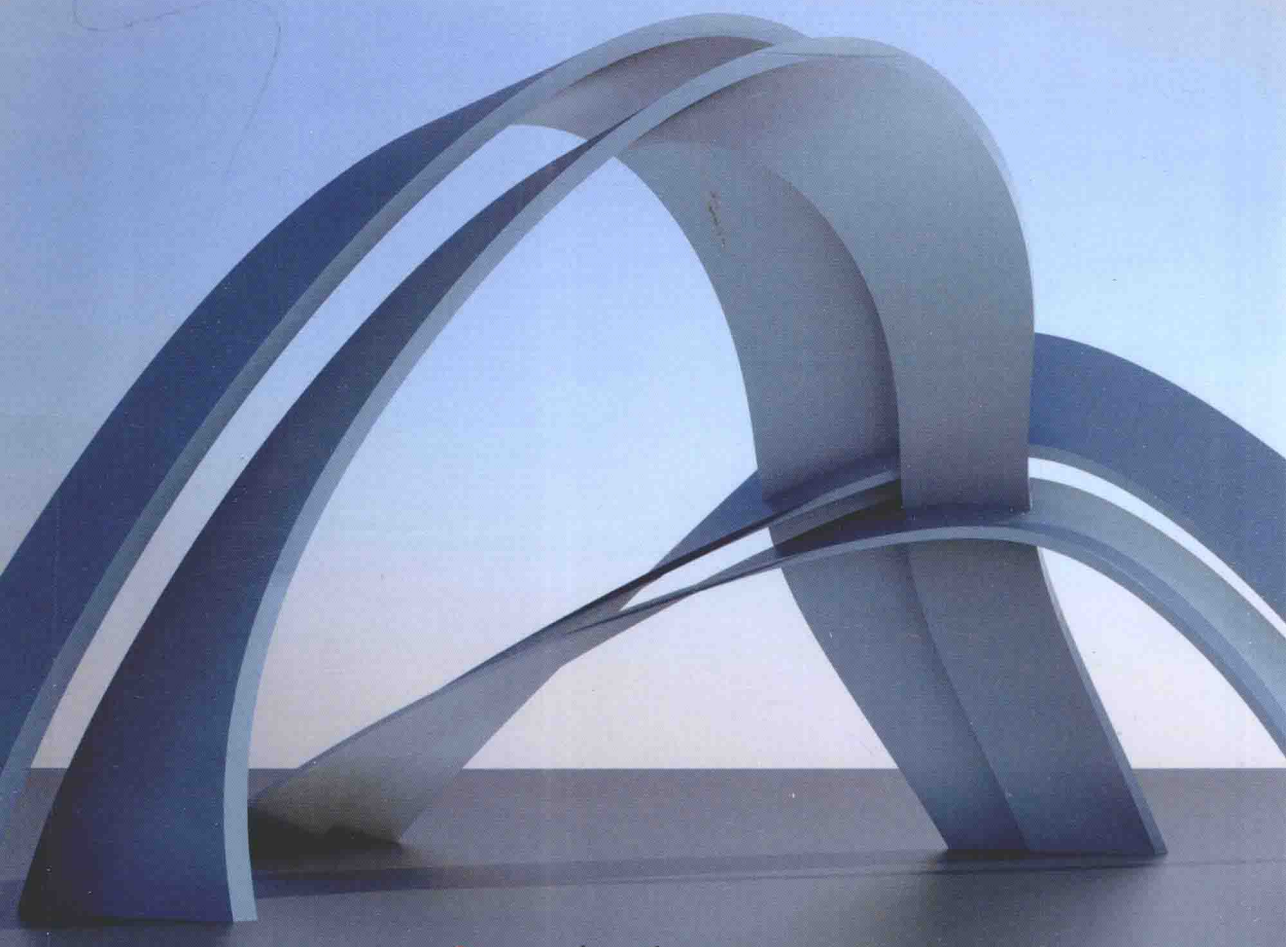
建筑信息模型BIM丛书
AUTODESK® REVIT®官方系列

AUTODESK® REVIT®
二次开发基础教程

主编　Autodesk Asia Pte Ltd



同济大学出版社
TONGJI UNIVERSITY PRESS

上架建议：计算机·建筑·土木

ISBN 978-7-5608-5962-0

定价：78.00 元

筑信息模型BIM丛书
Autodesk® Revit®官方系列

Autodesk® Revit® 二次开发基础教程

主　　编　Autodesk Asia Pte Ltd

编委会成员　（按姓氏笔画排序）

卢石碧　李健梅　李雪松
杨　俊　杨　涛　郑洁燕
戴　亮

同济大学出版社
TONGJI UNIVERSITY PRESS

内 容 提 要

本书由 Autodesk 公司的资深 Revit API 开发工程师和测试工程师编写，是国内首部介绍 Revit API 的权威书籍。以 Revit 2015 作为平台，由浅入深地介绍了 Revit API 的基础知识、开发工具以及相关资源，并搭配了大量的实例代码、图片和表格，方便读者更好的理解。本书内容是 Revit 二次开发基础类教程，共有 15 章，主要从以下几个方面来构建：功能（事件、界面、宏），类层次（应用类、文档类、元素、族等），不同专业（建筑、结构、MEP 各专业相关的 API）。

本书适用于想要使用 Revit API 进行编程的初学者。根据本书的 Revit API 知识，读者可以搭建起二次开发环境，并开发 Revit 插件来对 Revit 的数据进行读取、创建、修改、导入导出等；也可以通过 API 和. NET 技术来创建丰富的用户交互界面，提供更好的用户体验；通过扩展 Revit 本身不具备的功能，使得 Revit 和其他软件平台进行交互，实现数据验证、检查以及操作自动化，极大地提高数据利用率以及设计效率。

图书在版编目(CIP)数据

Autodesk® Revit® 二次开发基础教程/欧特克（中国）软件研发有限公司主编. —上海：同济大学出版社，2015. 8

（建筑信息模型 BIM 丛书. Autodesk® Revit® 官方系列）

ISBN 978-7-5608-5962-0

Ⅰ. ①A… Ⅱ. ①欧… Ⅲ. ①建筑设计—计算机辅助设计—应用软件—教材 Ⅳ. ①TU201. 4

中国版本图书馆 CIP 数据核字（2015）第 201867 号

Autodesk® Revit® 二次开发基础教程

欧特克（中国）软件研发有限公司 主编

责任编辑 赵泽毓 **助理编辑** 张富荣 **责任校对** 徐春莲 **封面设计** 陈益平

出版发行 同济大学出版社 www. tongjipress. com. cn

（地址：上海市四平路 1239 号 邮编：200092 电话：021-65985622）

经 销 全国各地新华书店

印 刷 同济大学印刷厂

开 本 787 mm×1 092 mm 1/16

印 张 17. 75

印 数 1—2 100

字 数 443 000

版 次 2015 年 8 月第 1 版 2015 年 8 月第 1 次印刷

书 号 ISBN 978-7-5608-5962-0

定 价 78. 00 元

本书若有印装质量问题，请向本社发行部调换 版权所有 侵权必究

前　言

众所周知，BIM(Building Information Modeling，建筑信息模型)的概念在建筑领域已深入人心，发展异常迅猛，尤其是当 BIM 被明确写入建筑业发展“十二五”规划并继续列入住建部、科技部“十三五”相关规划之后，BIM 发展趋势更是势不可挡。

然而，目前没有哪一款软件能够独自做到 BIM 全生命周期信息管理和共享，将来或许也不会出现。那么，应用程序编程接口(API)就成为打通各款软件信息交流的重要工具。

Autodesk® Revit®(以下简称 Revit)作为 BIM 软件的领跑者，除了自身功能强大之外，也提供丰富的 API，进一步奠定了 Revit 平台级软件地位。借助于 API 我们不用担心 Revit 建模之后的信息不能被其他软件所使用，借助于 API 我们可以把琐碎的建模工作自动化，借助于 API 我们可以把其他软件的功能集成或连接到 Revit 中来，在一个平台上就能完成几乎所有工作。

由于国内 Revit 二次开发一直没有相关书籍，市场呼声较高，而作为开发者在 Revit 上工作了 9～10 年的编者，也一直有这个想法，并最终成立编委会，历时一年完成了本书。

本书适合作为 Revit API 的初学者，所以涵盖的内容比较基础，目的是让新手可以尽快了解 Revit API 的构架，融入到 Revit 二次开发的队伍中来。

本书涵盖 Revit API 框架以及建筑、结构和水暖电的 API 等，由浅入深，第 1—3 章介绍概念，第 4 章是建筑建模，第 5—9 章涉及稍微中级一点内容，包括注释、几何、族、视图和事件，第 10 章介绍 UI 对应的 API，第 11—13 章涵盖结构建模、材料和水暖电建模，最后第 14 和 15 章介绍宏和其他语言(除 C# 以外的)调用 API。

本书的编委会成员都是资深的 Revit API 开发工程师和测试工程师，均在 Revit 上工作了有 9～10 年时间。第 1 章由郑洁燕和卢石碧编写，第 2 章由郑洁燕和杨涛编写，第 3 章由李健梅和卢石碧编写，第 4 章由卢石碧编写，第 5、8、9、14、15 章由杨涛编写，第 6 章由李健梅编写，第 7 章和第 12 章由李雪松编写，第 10 章由杨俊和郑洁燕编写，第 11 章由杨俊编写，第 13 章由戴亮和卢石碧编写。

本书也参考了 Revit wiki 上面的 Revit API Developer's Guide(http://help.autodesk.com/view/RVT/2015/ENU/?guid=GUID-F0A122E0-E556-4D0D-9D0F-7E72A9315A42)的部分内容，本书内容和上面不尽相同，有兴趣的读者，也可以通过以上站点学习更多 Revit API 相关的知识。

欢迎关注我们的新浪微博“Revit 二次开发”(http://weibo.com/revit-developers),将会有更多资源发布。另外,对本书有任何问题,意见或建议,也可以通过微博联系我们。

最后,衷心感谢三位领导——开发经理黄腾香、刘培珺和产品经理周杨海的鼎力支持,还有和我们一起工作的同事们的理解。谢谢编委会各成员的不懈努力与奉献,在百忙之中抽出时间做研究、测试并编写修改本书,这在一定程度上也耽误了和家人朋友的相处,所以也特别感谢编委会的家属和朋友的理解、支持与关怀。

编委会

2015 年 3 月

Guide　　微博

目　录

前　言

第 1 章　Revit® API 概述 …… 1
1.1　了解 Revit® 和 Revit® API …… 1
1.2　Revit® API 可以做什么 …… 1
1.3　使用 Revit® API 的准备工作 …… 2
1.4　网上资源 …… 2
1.5　开发工具 …… 3
1.5.1　Visual Studio 2012 …… 3
1.5.2　RevitSDK …… 3
1.5.3　ILSpy …… 3
1.5.4　RevitLookup …… 3
1.5.5　AddinManager …… 4

第 2 章　Revit® API 基础 …… 5
2.1　外部命令和外部应用(IExternalCommand/IExternalApplication) …… 5
2.1.1　外部命令 …… 5
2.1.2　外部应用 …… 10
2.1.3　数据库(DB)级别的外部应用 …… 10
2.1.4　注册 …… 11
2.1.5　属性 …… 13
2.2　Revit®应用类和文档类(Application/Document) …… 14
2.2.1　Revit®应用类 …… 15
2.2.2　Revit®文档类 …… 16
2.2.3　Revit®文档和文件管理 …… 18
2.2.4　类别(Category) …… 20
2.2.5　打印 …… 22
2.3　事务处理 (Transaction) …… 23
2.3.1　事务简介 …… 23
2.3.2　事务类 …… 23
2.3.3　在事件中使用事务 …… 27
2.4　实战示例 …… 27
2.4.1　实战 1：“Hello Revit” …… 28
2.4.2　实战 2：将“Hello Revit”添加到选项卡 …… 32

2.4.3 实战 3：从 Revit® 中选取元素 …… 34
2.4.4 实战 4：通过过滤器取到元素 …… 37

第 3 章 元素(Element) …… 39
3.1 元素基础 …… 39
3.1.1 相关类图 …… 39
3.1.2 如何获取元素 …… 40
3.1.3 获取和修改元素参数 …… 43
3.1.4 重要的属性和方法 …… 47
3.1.5 元素、族、族类型、族实例 …… 48
3.1.6 族、族类型和族实例的关系 …… 49
3.1.7 从族创建族实例 …… 49
3.1.8 元素的创建 …… 51
3.2 元素编辑 …… 52
3.2.1 移动 …… 53
3.2.2 旋转 …… 55
3.2.3 镜像 …… 57
3.2.4 删除 …… 58
3.2.5 组合 …… 59
3.2.6 线型阵列和圆弧型阵列 …… 60
3.2.7 族编辑 …… 61
3.3 元素过滤器(ElementFilters) …… 69
3.3.1 收集器(FilteredElementCollector) …… 69
3.3.2 元素过滤器的层次 …… 70
3.3.3 内建过滤器(Built-In filters) …… 71
3.3.4 元素快速过滤器(ElementQuickFilter) …… 72
3.3.5 元素慢速过滤器(ElementSlowFilter) …… 75
3.3.6 元素逻辑过滤器(ElementLogicalFilter) …… 78

第 4 章 建筑建模 …… 81
4.1 标高和轴网(Level/Grid) …… 81
4.1.1 标高 …… 81
4.1.2 轴网 …… 82
4.2 宿主元素 (HostObject) …… 83
4.2.1 类图 …… 83
4.2.2 获取复合结构(CompoundStructure) …… 84
4.2.3 获取宿主上的元素 …… 84
4.2.4 获取宿主的面 …… 84
4.2.5 创建墙(Wall) …… 85

4.2.6　创建楼板(Floor) …… 88
4.2.7　天花板和屋顶(Ceiling/Roof) …… 89
4.3　族实例 (FamilyInstance) …… 92
4.3.1　族实例的宿主 (Host) …… 92
4.3.2　旋转实例 (Rotation) …… 92
4.3.3　门和窗 (Door/Window) …… 92
4.3.4　翻转门窗 (Flip) …… 92
4.3.5　子构件和父构件 (SubComponent/SuperComponent) …… 93
4.4　族实例(FamilyInstance)的创建 …… 93
4.4.1　创建独立的族实例 …… 93
4.4.2　创建基于宿主的族实例 …… 94
4.4.3　创建基于标高的族实例 …… 95
4.4.4　创建基于标高的线形族实例 …… 95
4.4.5　创建基于视图的二维族实例 …… 96
4.4.6　创建基于视图的二维线形族实例 …… 97
4.4.7　创建基于面的族实例 …… 98
4.4.8　创建基于面的线形族实例 …… 99
4.4.9　批量创建族实例 …… 99
4.5　房间和面积(Room/Area) …… 101
4.5.1　房间、面积、标记(Room/Area/Tag) …… 101
4.5.2　创建房间 …… 101
4.5.3　创建面积 …… 103
4.5.4　房间边界 …… 104
4.5.5　平面拓扑(PlanTopology) …… 106
4.5.6　房间和实例 …… 107
4.6　线元素 (CurveElement) …… 108
4.6.1　获取模型线的几何信息 …… 109
4.6.2　获取模型线的工作平面 …… 109
4.6.3　获取和修改模型线样式 …… 109
4.6.4　创建模型线 …… 110
4.6.5　批量创建模型线 …… 111
4.7　洞口(Opening) …… 111
4.7.1　类图 …… 111
4.7.2　获取洞口的边界 …… 111
4.7.3　其他属性 …… 111
4.7.4　创建洞口 …… 112

第 5 章　注释 (Documentation) …… 113
5.1　尺寸标注(Dimension) …… 113

5.1.1 尺寸标注类型 …… 113
5.1.2 尺寸标注文字 …… 114
5.1.3 锁定尺寸标注 …… 115
5.1.4 尺寸标注的参照 …… 115
5.1.5 基线标注和同基准尺寸 …… 115
5.1.6 尺寸标注的创建 …… 116
5.1.7 高程点标注 …… 119
5.2 文字注释(Text) …… 120
5.2.1 文字注释的属性 …… 120
5.2.2 文字注释的创建 …… 121
5.3 详图(Detail) …… 121
5.3.1 详图线 …… 121
5.3.2 区域 …… 121
5.4 标记(Tag) …… 122
5.4.1 类别标记和材质标记 …… 123
5.4.2 注释记号(Keynote) …… 123
5.4.3 房间、空间和面积标记 …… 123
5.4.4 符号(Symbol) …… 124

第 6 章 几何 (Geometry) …… 126
6.1 概述 …… 126
6.2 实战:获取一堵墙的几何数据 …… 126
6.2.1 创建几何选项 …… 126
6.2.2 获取面和边 …… 127
6.3 几何基元类 …… 127
6.3.1 几何实例(GeometryInstance) …… 128
6.3.2 网格(Mesh) …… 130
6.3.3 实体(Solid) …… 131
6.4 几何辅助类 …… 132
6.4.1 几何变换(Transform) …… 133
6.4.2 几何引用(Reference) …… 135
6.4.3 几何选项(Options) …… 136
6.4.4 BoundingBoxXYZ …… 137
6.4.5 BoundingBoxUV …… 140
6.5 几何集合类 …… 141
6.6 实战:获取一根梁的几何数据 …… 141

第 7 章 族(Family) …… 143
7.1 族简介 …… 143

7.2 相关的主要 API 类 …… 143
7.3 族类型和族参数的管理 …… 143
7.3.1 获取管理类(FamilyManager) …… 143
7.3.2 创建和编辑族类型 …… 144
7.3.3 族类型的创建 …… 144
7.3.4 创建和编辑族参数 …… 145
7.4 几何图元的管理 …… 154
7.4.1 拉伸 …… 154
7.4.2 融合 …… 156
7.4.3 旋转 …… 159
7.4.4 放样 …… 161
7.4.5 放样融合 …… 163
7.4.6 图元连接 …… 165
7.4.7 小结 …… 165
7.5 几何图元的可见性管理 …… 165
7.6 编辑族和载入族 …… 166
7.6.1 编辑族 …… 166
7.6.2 载入族 …… 166
7.7 其他 …… 167
7.7.1 族类别和族参数 …… 167
7.7.2 概念体量 …… 167

第 8 章 视图(Views) …… 168
8.1 概述 …… 168
8.1.1 视图处理 …… 168
8.1.2 视图类型 …… 169
8.1.3 视图中元素的可见性 …… 171
8.1.4 创建和删除视图 …… 172
8.2 三维视图(View3D) …… 173
8.2.1 透视三维视图(Perspective View3D) …… 174
8.2.2 正交三维视图(Orthographic View3D) …… 175
8.2.3 三维视图剖面框(Section Box) …… 176
8.2.4 三维视图的旋转 …… 177
8.3 平面视图(ViewPlan) …… 177
8.4 绘图视图(ViewDrafting) …… 178
8.5 剖面视图(ViewSection) …… 178
8.6 参照详图索引视图和详图视图 …… 179
8.7 图纸视图(ViewSheet) …… 180
8.8 明细表(ViewSchedule) …… 181

第 9 章　事件(Events) …… 185
9.1　事件简介 …… 185
9.2　注册和注销事件 …… 185
9.3　可取消的事件 …… 187
9.4　数据库事件 …… 187
9.5　界面事件 …… 189
9.6　闲置事件(IdlingEvent) …… 190
9.7　外部事件(ExternalEvent) …… 191

第 10 章　功能区扩展(Ribbon UI) …… 195
10.1　基础介绍 …… 195
10.2　选项卡页(RibbonTab) …… 196
10.3　面板(RibbonPanel) …… 196
10.4　命令按钮(PushButton) …… 198
10.5　下拉按钮(PulldownButton) …… 199
10.6　下拉记忆按钮(SplitButton) …… 199
10.7　下拉组合框(ComboBox) …… 201
10.8　下拉组合框的可选项(ComboBoxMember) …… 201
10.9　选择按钮组和切换按钮(RadioButtonGroup/ToggleButton) …… 202
10.10　文本框(TextBox) …… 204
10.11　Revit®风格的任务对话框(TaskDialog) …… 206
10.11.1　任务对话框的控件 …… 206
10.11.2　创建和显示自定义任务对话框 …… 208
10.11.3　任务对话框执行结果 …… 209

第 11 章　Revit®结构建模 …… 211
11.1　结构模型元素 …… 211
11.1.1　基本元素 …… 211
11.1.2　荷载 …… 212
11.1.3　边界条件 …… 214
11.2　分析模型(AnalyticalModel) …… 216
11.2.1　概述 …… 216
11.2.2　分析模型的几何信息 …… 217
11.2.3　分析模型曲线的几何引用(Reference) …… 218
11.2.4　分析模型参数相关的 API …… 220
11.2.5　支撑信息(AnalyticalModelSupport) …… 223
11.2.6　编辑分析模型 …… 227
11.2.7　分析链接(AnalyticalLink) …… 229

第 12 章　材料(Material) …… 232
12.1　材料简介 …… 232
12.2　材料的标识 …… 232
12.3　材料的图形信息 …… 232
12.4　材料的外观信息 …… 233
12.4.1　外观元素 …… 233
12.4.2　访问外观属性集合 …… 234
12.5　材料的物理和热度信息 …… 236
12.6　材料的设置 …… 238
12.6.1　组类别的预定义材料 …… 238
12.6.2　墙等复合结构的材料设置 …… 238
12.6.3　楼梯等复杂结构的材料设置 …… 239
12.6.4　构件的材料设置 …… 239
12.6.5　面的材料设置 …… 239

第 13 章　水暖电建模 …… 241
13.1　风管/管道 (Duct/Pipe) …… 241
13.1.1　风管(Duct) …… 241
13.1.2　风管属性和方法 …… 241
13.1.3　创建一般风管 …… 241
13.1.4　创建软风管 …… 243
13.1.5　创建风管占位符 …… 244
13.1.6　管道(Pipe) …… 244
13.2　电气连接件(Connector) …… 244
13.3　水暖电模型(MEPModel) …… 245
13.4　水暖电系统(MEPSystem) …… 247
13.4.1　水暖电系统的创建 …… 248
13.4.2　水暖电系统属性和方法 …… 248
13.4.3　创建风管系统 …… 248
13.5　水暖电设置 …… 250
13.5.1　风管设置 …… 251
13.5.2　管道设置 …… 252
13.5.3　电气设置 …… 252
13.6　空间和分区(Space/Zone) …… 252

第 14 章　宏(Macro) …… 255
14.1　Revit®宏简介 …… 255
14.1.1　宏管理器 …… 255
14.1.2　宏和模块 …… 256

14.1.3 应用程序级和文档级模块 …… 256
14.1.4 Revit® 宏实现语言 …… 257
14.1.5 宏项目文件位置 …… 257
14.2 Revit® 宏开发的基本工作流程 …… 257
14.2.1 创建模块 …… 257
14.2.2 创建宏 …… 258
14.2.3 宏示例代码 …… 260
14.3 修改与删除模块和宏 …… 263
14.3.1 模块 …… 263
14.3.2 宏 …… 263
14.4 在宏管理器中运行宏 …… 263
14.5 调试宏 …… 263
14.6 宏安全性 …… 264
14.7 标准 Revit® API 与 Revit® 宏使用的 API 的区别 …… 265

第 15 章 其他语言（VB.NET，C++/CLI，F#） …… 266
15.1 VB.NET …… 266
15.2 C++/CLI …… 267
15.3 F# …… 269

第 1 章　Revit® API 概述

1.1　了解 Revit®和 Revit® API

Autodesk® Revit®是为建筑信息模型(Building Information Modeling，BIM)而设计的系列软件，包括 Revit® Architecture，Revit® Structure，Revit® MEP，Revit® OneBox 以及 Revit®LT 等一系列产品，分别为建筑、结构、设备(水、暖、电)等不同专业提供 BIM 解决方案。

所有 Revit®产品都是参数化的。“参数化”是指模型的所有元素之间的关系，这些关系可实现 Revit®提供的协调和变更管理功能。这些关系可以由软件自动创建，也可以由设计者在项目开发期间创建。

这与 CAD(Computer-Aided Design)非常相似，不同之处是 Revit®通过 3D 模型来建模，而不是通过草图。在 Revit®中，可以直接将现实生活中对应的元素(比如柱子、墙等)加到模型中。当模型创建完毕，可以生成对应的各种图纸、二维视图、三维视图以及明细表等。

在 Revit®模型中，所有的图纸、二维视图和三维视图以及明细表都是同一个基本建筑模型数据库的信息表现形式。在图纸视图和明细表视图中操作时，Revit®将收集有关建筑项目的信息，并在项目的其他所有表现形式中同步该信息。Revit®参数化修改引擎可自动协调在任何位置(模型视图、图纸、明细表、剖面和平面等)进行的修改。

Autodesk® Revit®系列的所有产品都提供 API(Application Programming Interface)，高级用户和第三方开发者能够通过 Revit® API 将他们的应用程序集成到 Autodesk® Revit®系列产品中。Autodesk® Revit®系列的三个产品的 API 非常相似，因此它们被集成到一个总的 API 包：Autodesk® Revit® API(简称 Revit® API)。

Revit® API 允许使用者通过任何与. NET 兼容的语言来编程，这些语言有 Visual Basic. NET、C#、C++/CLI、F#等。

建议在开始使用 Revit® API 之前，先熟悉 Revit®系列产品及其功能。这样可以帮助用户：

- 设计与 Revit®界面和命令保持一致的插件；
- 快速有效地理解和掌握 API 的类以及类里面的成员；
- 如果用户对 Revit®或者 BIM 不太熟悉，可以前往 Revit®产品中心 (http://www.autodesk.com.cn/products/autodesk-revit-family/overview)了解更多信息。

1.2　Revit® API 可以做什么

使用 API 之前我们有必要了解 API 能做什么。以下是使用 Revit® API 可以做到的事情：

- 访问模型的图形数据；
- 访问模型的参数数据；

- 创建、修改、删除模型元素；
- 创建插件来完成对 UI 的增强；
- 创建插件来完成一些对重复工作的自动化；
- 集成第三方应用来完成诸如连接到外部数据库、转换数据到分析应用等；
- 执行一切种类的 BIM 分析；
- 自动创建项目文档。

1.3 使用 Revit® API 的准备工作

在使用 Revit® API 和本书之前，需要具备以下条件：

- 安装 Revit® 系列产品，了解 Revit® 系列产品的功能和使用。Revit® API 在安装 Revit® 系列产品时也会自动安装。
- 了解至少一种符合公共语言规范的编程语言，如 C#。更多关于公共语言规范，请参见：http://msdn.microsoft.com/zh-cn/library/12a7a7h3(v=vs.90).aspx。
- 安装 Microsoft.NET Framework 4.5。
- 安装支持 Microsoft.NET Framework 4.5 的 IDE（推荐使用 MircoSoft Visual Studio2012 或更高版本）。用户也可以选择使用 Revit® 宏功能来实现自己的插件。想了解更多 Revit® 宏功能，请参见本书“第 14 章宏(Macro)”。

1.4 网上资源

下面列出了获取 Revit® 或 Revit® API 相关资源的网络途径。

1. Autodesk 资源
- Autodesk ADN 官网：http://www.autodesk.com/developrevit
- Revit 2015 中文帮助，包含英文 RevitAPI 帮助：http://help.autodesk.com/view/RVT/2015/CHS/
- Revit 官方论坛：http://forums.autodesk.com/t5/revit/ct-p/2003
- Revit 产品系列官网（中文）：http://www.autodesk.com.cn/products/autodesk-revit-family/overview
- Autodesk BIM 解决方案（英文）：http://usa.autodesk.com/building-information-0modeling/

2. 外部资源
- Revit 二次开发微博：http://www.weibo.com/revitdevelopers
- Revit 二次开发博客 1：http://blog.csdn.net/lushibi
- Revit 二次开发博客 2：http://blog.csdn.net/joexiongjin
- EaBIM 中文论坛：http://www.eabim.net/
- 中国 BIM 论坛：http://www.bim123.com/
- Revit 英文论坛：http://www.augi.com/community/autodesk-revit

3. 编程知识相关资源
- http://msdn.microsoft.com/en-us/vstudio
- http://www.codeguru.com

- http://www.devx.com

读者可通过扫描如下二维码进入网上资源。



1.5　开发工具

1.5.1　Visual Studio 2012

Revit® 2015 是基于.NET 4.5 的运行环境的，所以，需要使用 Visual Studio 2012 或更高版本，用.NET 4.5 环境来编译。

1.5.2　RevitSDK

RevitSDK 包含 RevitAPI 的帮助文档以及带源代码的例子，下载地址是：http://www.autodesk.com/developrevit，找到 Revit 2015 SDK 后可进行下载。

1.5.3　ILSpy

ILSpy 是一个免费的.NET 反射工具，可以用来查看一个.NET Assembly 的类和函数，软件界面见图 1-1。开发过程中经常利用它来查看类和函数，以及类之间的关系，比如继承关系，或者搜索开发者想要的内容。更详细的使用介绍可以参见工具官网。

下载链接：百度或者谷歌搜索即可。

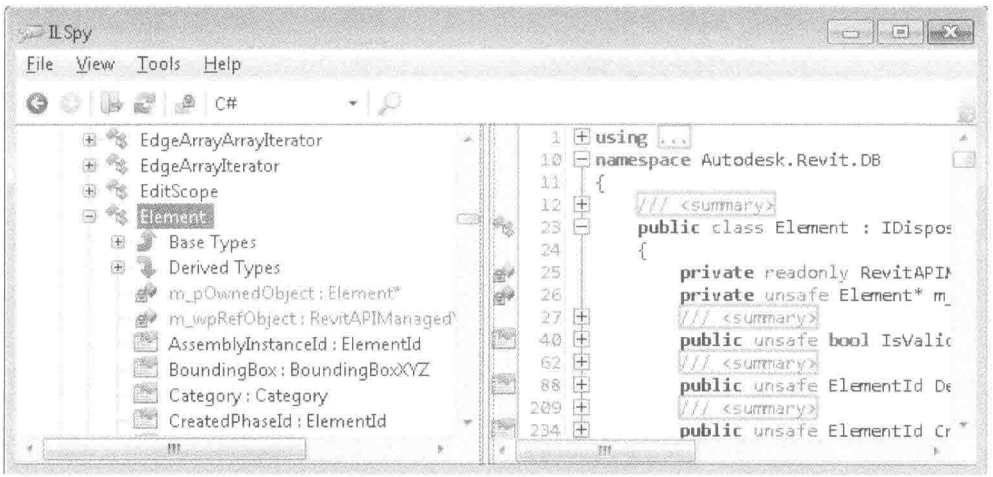


图 1-1　ILSpy 界面

1.5.4　RevitLookup

RevitLookup 是 Autodesk 开发的插件，见图 1-2，不用写代码就可以直观地看到 API 的对象。它包含在 RevitSDK.zip 压缩包中，可以拿到源代码后自己编译。SDK 的下载地址是：http://usa.autodesk.com/adsk/servlet/index?id=2484975&siteID=123112。

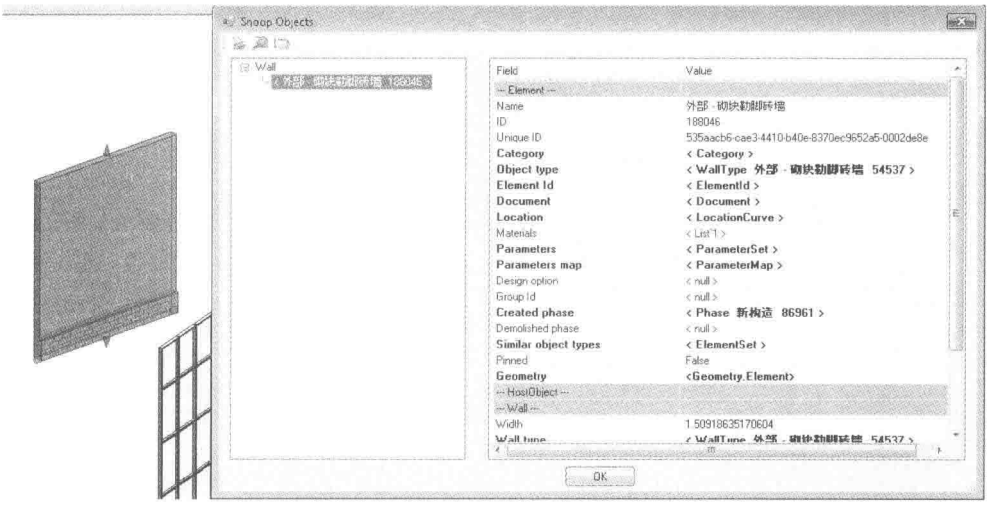


图 1-2　使用 RevitLookup 查看墙的信息

1.5.5　AddinManager

AddinManager 也是 Autodesk 的官方插件，用来加载 Revit® 的插件，界面见图 1-3。它的优点是不用重启 Revit® 就可以修改插件代码并再次加载和运行。它也包含在 RevitSDK 中。

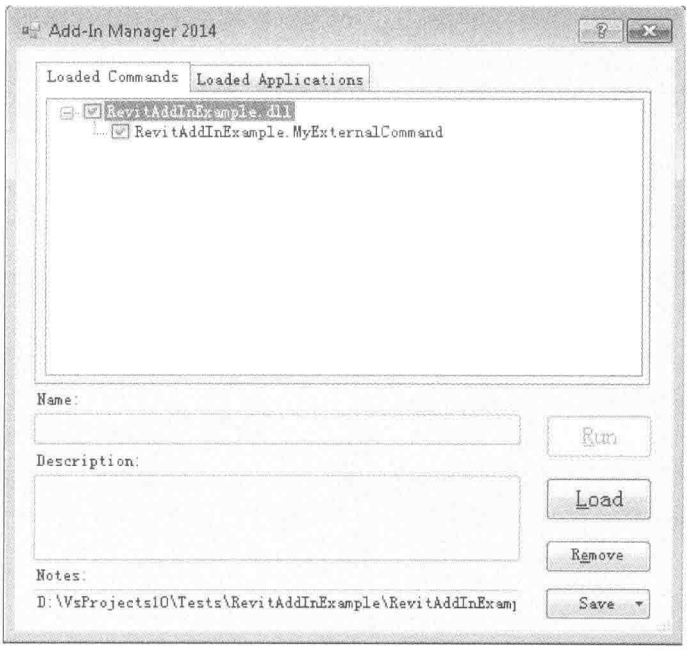


图 1-3　AddInManager 界面

第 2 章　Revit® API 基础

本章主要介绍使用 Revit® API 的基础知识，包括实现插件必须继承的接口 IExternalCommand/IExternalApplication、用于保存 Revit® 中数据的 Application 和 Document、Revit®中如何进行事务处理。本书 2.4 节的 4 个实战示例可以作为理解 Revit®以及 Revit® API 的起点。

2.1　外部命令和外部应用(IExternalCommand/IExternalApplication)

Revit® API 是建立在 Revit®产品的基础之上的。它是一个类库，需要在 Revit®运行时才能够工作。通过这套强大的 Revit® API，可以添加用户基于 Revit® API 开发的插件来扩展和增强 Revit®的功能和应用。

如图 2-1 所示，Revit® Architecture、Revit® Structure 以及 Revit® MEP 分别为建筑、结构和设备(水、暖、电)专业提供 BIM 解决方案，Revit® API 建立在这三个产品之上，覆盖 Revit®所有产品的功能。

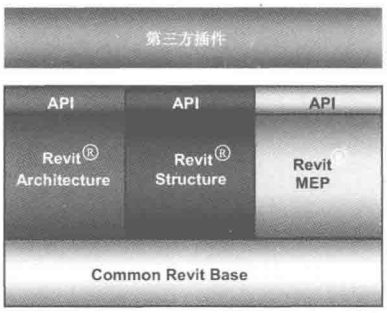


图 2-1　Revit®系列产品与 API

Revit® API 提供了一套机制和规范来扩展 Revit®的功能。

Revit® API. dll 程序集包含了访问 Revit®中 DB 级别的 Application、Document、Element 以及 Parameter 的方法，也包含了 IExternalDBApplication 接口和其他相关接口。

Revit® APIUI. dll 程序集包含了所有操作和定制 Revit® UI 的接口，包括：

- IExternalCommand 相关接口；
- IExternalApplication 相关接口；
- Selection 选择；
- 菜单类 RibbonPanel，RibbonItem 以及其子类；
- TaskDialogs 任务对话框。

想通过 Revit® API 来访问和扩展 Revit®，需要在用户自己的插件中实现特殊的接口。这些接口包括 IExternalCommand、IExternalApplication 和 IExternalDBApplication。本章将详细介绍这些接口以及它们的注册方式。

2.1.1　外部命令

插件开发者可以通过 IExternalCommand 来添加自己的应用。Revit®通过. addin 文件来识别和加载外部插件。

1) 基本原理

如果 Revit®没有其他的命令在运行或者没有处于编辑模式，那么已经注册了的外部命令(ExternalCommand)就可以被激活。一旦插件被选中，外部命令对象将被创建出来，并且

执行 Execute 函数。执行完毕后，外部命令对象被销毁。因此，在两次外部命令执行之间，数据不能被保持在对象之中。当然，用户可以通过其他方式来实现在 Revit® 中存储数据。

图 2-2 描述了外部命令是如何被调用的。

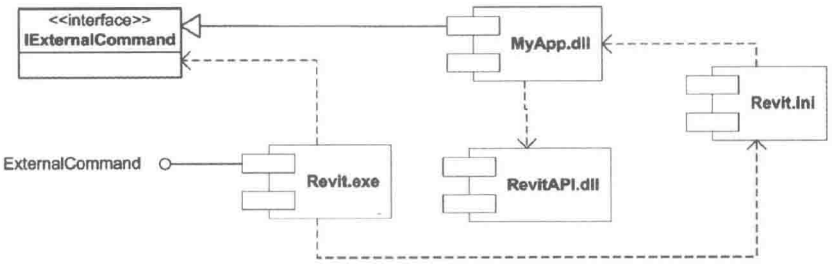


图 2-2 调用外部命令

外部命令被集成到 Revit® 之后，可以通过外部工具菜单(图 2-3)和自定义菜单项(图 2-4)两种方式来触发外部命令。



图 2-3 外部工具菜单

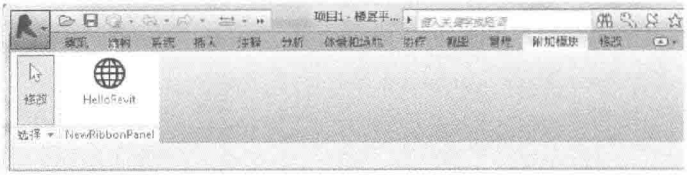


图 2-4 自定义菜单项

2) IExternalCommand

IExternalCommand 是 Revit® API 用户通过外部命令来扩展 Revit® 时必须在外部命令中实现的接口。IExternalCommand 接口只有一个抽象函数 Excute，重载这个函数来实现 IExternalCommand。Excute 函数将作为外部命令的主函数被调用，代码片段 2-1 是它的具体定义。

代码片段 2-1：外部命令中 Excute 函数的定义

```
public interface IExternalCommand
{
  public Autodesk.Revit.UI.Result Execute(
Autodesk.Revit.UI.ExternalCommandData commandData,
ref string message,
Autodesk.Revit.DB.ElementSet elements)
}
```

Execute 函数有三个参数：输入参数 commandData（ExternalCommandData）、输出参数 message（String）、输出参数 elements（ElementSet）。

下面将详细介绍这三个参数的用法。

（1）输入参数 commandData（ExternalCommandData）

ExternalCommandData 对象包含了外部命令所需的 Application 以及一些视图的引用。在外部命令中，所有的 Revit® 的数据都可以通过这个参数直接或间接地取到。

代码片段 2-2 演示如何从 commandData 中取到 Revit® 的 Application 和 Document。

代码片段 2-2：从 commandData 中取到 Document

```
UIApplication uiApplication = commandData.Application;
Application application = uiApplication.Application;
UIDocument uiDocument = uiApplication.ActiveUIDocument;
Document document = uiDocument.Document;
```

表 2-1 列出的是 ExternalCommandData 中的属性。

表 2-1　　ExternalCommandData 中的属性

属　性	说　明
Application(Autodesk. Revit. UI. UIApplication)	用于表示适用于当前外部命令的 Application 对象（详见 2.2 Revit® 应用类和文档类）
View(Autodesk. Revit. DB. View)	用于表示当前外部命令起作用的视图对象
JournalData(IDictionary<String,String>)	用于在 Revit Journal 文件中读写数据的数据字典，主要应用于做自动化测试时与外部命令做交互使用

（2）输出参数 message（String）

外部命令可以通过这个参数来返回执行过程中的错误信息。这个参数作用于整个外部命令的执行过程，用户可以在外部命令执行过程中的任何时候给这个信息设值或者追加信息。当外部命令的 Execute 函数返回 Autodesk. Revit. UI. Result. Failed 或者 Autodesk. Revit. UI. Result. Canceled，这个错误信息将被显示在 UI 上。

代码片段 2-3 演示如何使用 message 这个参数。

代码片段 2-3：使用 message 参数

```
public class command: IExternalCommand
{
    public Result Execute(
ExternalCommandData commandData,
ref string message,
ElementSet elements)
    {
      message = "message test";
      return Result.Failed;
    }
}
```

执行结果如图 2-5 所示。

(3) 输出参数 elements (ElementSet)

当外部命令返回 Autodesk. Revit. UI. Result. Failed 或者 Autodesk. Revit. UI. Result. Canceled 并且 message 参数不为空的时候，错误或者警告对话框会弹出来，点击上面的显示按钮，elements 参数中的元素将被高亮显示。

代码片段 2-4 演示如何使用 elements 这个参数。



图 2-5 执行结果

代码片段 2-4：使用 elements 参数

```
public Result Execute(ExternalCommandData commandData, ref string message, ElementSet
    elements)
{
  message = "Please take attention on the highlighted Walls!";
  //先从 UI 选取元素，然后执行该插件
  ElementSet elems = commandData.Application.ActiveUIDocument.Selection.Elements;
  foreach (Element elem in elems)
  {
      Wall wall = elem as Wall;
      if (null != wall)
      {
  elements.Insert(elem);
      }
  }
  return Result.Failed;
}
```

执行结果如图 2-6 所示。



图 2-6 执行结果

(4) 返回值

Execute 函数的返回值用于表示外部命令的执行状态，分为 Autodesk. Revit. UI. Result. Succeeded，Autodesk. Revit. UI. Result. Failed 和 Autodesk. Revit. UI. Result. Canceled 三种。如果返回值不为 Succeeded，那么 Revit® 会把外部命令所做的所有操作和修改撤销。

代码片段 2-5 演示 Revit® 如何处理不同的外部执行状态。

代码片段 2-5：外部命令中 Excute 函数的返回值

```
public Result Execute (ExternalCommandData commandData, ref string message, ElementSet
elements)
      {
        try
        {
            UIDocument uiDoc =  commandData.Application.ActiveUIDocument;
            Document doc =  uiDoc.Document;
            List< ElementId>  selectedElem =  new List< ElementId> ( );
            foreach(Element elem in uiDoc.Selection.Elements)
            {
              selectedElem.Add(elem.Id);
            }

            doc.Delete(selectedElem);

            TaskDialogResult result =  TaskDialog.Show(
              "Revit",
              "Yes to return succeeded and delete all selection,"+
              "No to cancel all commands.",
              TaskDialogCommonButtons.Yes|TaskDialogCommonButtons.No);

            if (TaskDialogResult.Yes = =  result)
            {
              return Result.Succeeded;
            }
            else if (TaskDialogResult.No = =  result)
            {
              elements =  uiDoc.Selection.Elements;
              message =  "Failed to delete selection.";
              return Result.Failed;
            }
            else
            {
              return Result.Cancelled;
            }
        }
        catch
        {
            message =  "Unexpected Exception is thrown out.";
            return Result.Failed;
        }

      }
```

执行结果如图 2-7 所示。

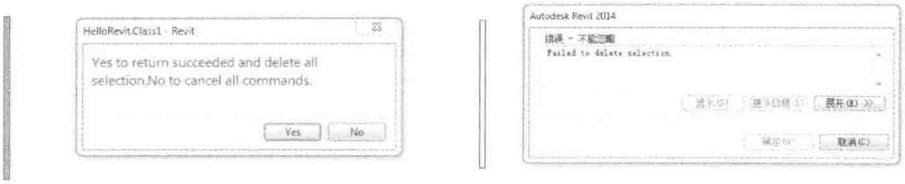


图 2-7　执行结果

2.1.2 外部应用

插件开发者同样可以通过实现 IExternalApplications 来添加自己的应用。Revit® 同样通过. addin 文件来识别和加载实现 IExternalApplication 的外部插件。

IExternalApplication 接口有两个抽象函数 OnStartup 和 OnShutdown。用户可以通过在实现了 IExternalApplication 的外部应用中重载 OnStartup 和 OnShutdown 函数，在 Revit® 启动和关闭的时候定制所需的功能。代码片段 2-6 是这两个函数的具体定义。

代码片段 2-6：IExternalApplication 接口定义

```
public interface IExternalApplication
  {
    Autodesk.Revit.UI.Result OnShutdown(UIControlledApplication application);
    Autodesk.Revit.UI.Result OnStartup(UIControlledApplication application);
  }
```

OnStartup 和 OnShutdown 函数的参数均为 UIControlledApplication 类型。

UIControlledApplication 类是一种特殊的应用类，它不提供访问 Revit® 文档的途径。因为 UIControlledApplication 只在 OnStartup 和 OnShutdown 函数范围内起作用，在这个作用域区间内用户是无法拿到 Revit® 文档并对其操作的。UIControlledApplication 类提供访问定制 UI 和注册事件的方法。代码片段 2-7 简单演示如何定制 UI。想了解更多定制 UI，请参见功能区 UI 扩展章节。

代码片段 2-7：使用 IExternalApplication 定制 UI

```
      public Autodesk.Revit.UI.Result OnStartup(UIControlledApplication application)
      {
//添加一个新的 Ribbon 面板
        RibbonPanel ribbonPanel =  application.CreateRibbonPanel("NewRibbonPanel");

        //在新的 Ribbon 面板上添加一个按钮
        //点击这个按钮，调用 2.4 节第一个实例
        PushButton pushButton =  ribbonPanel.AddItem(new PushButtonData("HelloRevit",
            "HelloRevit", @ "C:\Projects\HelloRevit\HelloRevit.dll", "HelloRevit.Class1"))
            as PushButton;
        return Result.Succeeded;
      }

      public Result OnShutdown(UIControlledApplication application)
      {
        //UI 定制不需要特别在 OnShutdown 方法中做处理
        return Result.Succeeded;
      }
```

2.1.3 数据库(DB)级别的外部应用

数据库(DB)级别的外部应用与一般外部应用不同的是，它没有提供任何访问 Revit® UI 的接口。它一般用于事件处理，插件开发者可以通过实现 IExternalDBApplication 来添加所需的 DB 基本的外部应用。

同 IExternalApplication 一样，IExternalDBApplication 接口也有两个抽象函数 OnStartup 和 OnShutdown。代码片段 2-8 是这两个接口的定义。

代码片段 2-8：IExternalDBApplication 接口定义

```
public interface IExternalDBApplication
{
Autodesk. Revit. DB. ExternalDBApplicationResultOnShutdown ( UIControlledApplication
application);
Autodesk. Revit. DB. ExternalDBApplicationResultOnStartup ( UIControlledApplication
application);
}
```

用户可以在 OnStartup 方法中注册所需的 DB 级别的事件。

2.1.4　注册

如果想在 Revit® 中调用插件，需要对插件进行注册。Revit® 通过后缀名为. addin 的文件(XML 格式)来实现注册。Revit® 会在启动时自动搜索下列目录中的. addin 文件并进行加载。

(1) 如果希望该插件只有当前用户可以使用，需要将文件放入：

① (Windows XP)C:\Documents and Settings\<user>\Application Data\Autodesk\Revit®\Addins\2015\

② (Window 7)C:\Users\<user>\AppData\Roaming\Autodesk\Revit\Addins\2015\

(2) 如果希望所有用户都可以使用，需要将文件放入：

① (Windows XP)C:\Documents and Setting\All Users\Application Data\Autodesk\Revit®\Addins\2015\

② (Window 7)C:\ProgramData\Autodesk\Revit\Addins\2015\

对于前三节中讲到的不同类型的插件，Revit® 提供了不同的注册格式和 XML 标签。

代码片段 2-9 是一个 ExternalCommand 的. addin 文件格式示例。

代码片段 2-9：ExternalCommand 的. addin 文件格式示例

```
< ? xml version= "1.0" encoding= "utf-8" standalone= "no"? >
< RevitAddIns>
  < AddIn Type= "Command">
    < Assembly> c:\MyProgram\MyProgram.dll< /Assembly>
    < AddInId> 76eb700a-2c85-4888-a78d-31429ecae9ed< /AddInId>
    < FullClassName> Revit.Samples.SampleCommand< /FullClassName>
    < Text> Sample command< /Text>
    < VisibilityMode> NotVisibleInFamily< /VisibilityMode>
    < VisibilityMode> NotVisibleInMEP< /VisibilityMode>
        < AvailabilityClassName > Revit. Samples. SampleAccessibilityCheck < /
        AvailabilityClassName>
    < LongDescription> < p> This is the long description for my command.< /p> < p> This is
    another descriptive paragraph, with notes about how to use the command properly.< /p>
    < /LongDescription>
```

```
    < TooltipImage> c:\MyProgram\Autodesk.jpg< /TooltipImage>
    < LargeImage> c:\MyProgram\MyProgramIcon.png< /LargeImage>
    < VendorId> ADSK< /VendorId>
    < VendorDescription> Autodesk, www.autodesk.com< /VendorDescription>
  < /AddIn>
< /RevitAddIns>
```

代码片段 2-10 是一个 ExternalApplication 的. addin 文件格式示例。

代码片段 2-10:ExternalApplication 的. addin 文件格式示例

```
< ? xml version= "1.0" encoding= "utf-8" standalone= "no"? >
< RevitAddIns>
  < AddIn Type= "Application">
    < Name> SampleApplication< /Name>
    < Assembly> c:\MyProgram\MyProgram.dll< /Assembly>
    < AddInId> 604B1052-F742-4951-8576-C261D1993107< /AddInId>
    < FullClassName> Revit.Samples.SampleApplication< /FullClassName>
    < VendorId> ADSK< /VendorId>
    < VendorDescription> Autodesk, www.autodesk.com< /VendorDescription>
  < /AddIn>
< /RevitAddIns>
```

代码片段 2-11 是一个数据库级别 ExternalApplication 的. addin 文件格式示例。

代码片段 2-11:数据库级别 ExternalApplication 的. addin 文件格式示例

```
< ? xml version= "1.0" standalone= "no"? >
< RevitAddIns>
  < AddIn Type= "DBApplication">
      < Assembly> c:\MyDBLevelApplication\MyDBLevelApplication.dll< /Assembly>
      < AddInId> DA3D570A-1AB3-4a4b-B09F-8C15DFEC6BF0< /AddInId>
      < FullClassName> MyCompany.MyDBLevelAddIn< /FullClassName>
      < Name> My DB-Level AddIn< /Name>
      < VendorId> ADSK< /VendorId>
      < VendorDescription> Autodesk, www.autodesk.com< /VendorDescription>
  < /AddIn>
< /RevitAddIns>
```

对多个插件注册,可以将它们放到同一个. addin 文件中。表 2-2 是. addin 文件中所提供的各种 XML 标签的介绍。

表 2-2　　addin 文件中的 XML 标签

XML 标签	描　述
Assembly	指定插件程序集的全路径。每个 ExternalCommand 和 ExternalApplication 都必须要有这个值
FullClassName	指定实现了 IExternalCommand 或者 IExternalApplication 接口的插件程序集的类名。注意需要包括完整的命名空间。每个 ExternalCommand 和 ExternalApplication 都必须要有这个值

续表

XML 标签	描　述
AddInId	指定一个全局的 GUID 值。每一个插件都需要有一个 GUID 值，并且在一个 Revit® 会话中，这个 GUID 值需要是唯一的。在插件中可以通过 UIApplication. ActiveAddInId 属性取到. addin 文件中对应插件的 GUID 值
Name	为插件指定一个名字。只有类型为 ExternalApplication 的插件需要指定这个值
Text	为插件指定一个在“附加模块”选项卡中显示的名字。类型为 ExternalCommand 的插件可以指定这个值
Description	为插件提供一段简短的描述，这段描述将作为按钮的提示文本。这个标签不是必须的，如不提供将默认显示名字作为其提示文本。只有类型为 ExternalCommand 的插件可以使用这个标签
VisibilityMode	为插件指定可见属性。如不指定将默认为在任何情况下都显示插件，包括没有 Active Document 的情况。只有类型为 ExternalCommand 的插件可以使用这个标签
Discipline	为插件指定在哪个产品可见。如不指定将默认为所有产品下都显示。只有类型为 ExternalCommand 的插件可以使用这个标签
LargeImage	为插件设置在外部工具下拉菜单中显示的大图标。如不设置将默认不显示图片。只有类型为 ExternalCommand 的插件可以使用这个标签
SmallImage	为插件设置在快速访问工具条中显示的小图标。只有类型为 ExternalCommand 的插件可以使用这个标签
LongDescription	为插件提供一段长的描述，这段描述将作为按钮的提示文本的扩展部分。只有类型为 ExternalCommand 的插件可以使用这个标签
TooltipImage	为插件的提示文本提供一个图片，当鼠标在这个外部命令上停留时显示。只有类型为 ExternalCommand 的插件可以使用这个标签
LanguageType	为插件设置语言种类。其值可以是 Revit® 支持的 11 种语言中的一种。Revit® 会在对应语言种类的资源包中为外部工具按钮加载对应的 Text，Description，LargeImage，LongDescription 以及 TooltipImage 等的本地化资源
AllowLoadInto ExistingSession	将该标签的值设成 false 将阻止 Revit® 不重启就自动加载新加入到. addin 文件里面的插件。如不设置，Revit® 默认不重启自动加载新加入到. addin 文件里面的插件
VendorId	该标签标明该插件的插件提供商名称。对于 ExternalCommand 和 ExternalApplication 都必须为该标签指定一个值。该标签的值需要遵循 Autodesk 插件提供商命名标准。请在 Autodesk 的 http://www. autodesk. com/symbreg 网址注册提供商名称
VendorDescription	为插件设置插件提供商的描述

2.1.5　属性

Revit® API 提供了多种属性方便用户配置自己的 ExternalCommand 和 External Application 的行为。

1) Transaction 属性

用户必须在实现 IExternalCommand 接口时指定外面命令使用的 TransactionMode 属性。其格式如代码片段 2-12 所示。

代码片段 2-12：外部命令中 Excute 函数的 Transaction 属性

```
[Autodesk. Revit. Attributes. Transaction (Autodesk. Revit. Attributes. TransactionMode.
Automatic)]
public class Class1: IExternalCommand
{
}
```

该属性没有默认值，用户必须显式地指定自己想要的值。该属性控制 Revit® API 框架如何在调用外面命令时处理其中的事务(Transaction)。如果想了解更多 Transaction，请参见“2.3 事务处理”。

下面为当前 Revit® API 支持的 Transaction 属性模式。

① TransactionMode. Automatic：Revit® 会在调用外部命令之前自动创建一个 Transaction，并且该 Transaction 会在外面命令结束时提交或者回滚(外部命令执行成功，提交 Transaction。外部命令出现异常，则回滚)。在该模式下，外部命令不能创建自己的 Transaction，但是可以创建 SubTransactions。

② TransactionMode. Manual：在该模式下 Revit® 不会自动创建一个 Transaction。如果用户需要修改 Revit® 模型，则需要在自己的外部命令中来创建自己的 Transaction，并且自己管理这个 Transaction 是提交还是回滚。

③ TransactionMode. ReadOnly：在该模式下，外部命令不能有任何的 Transaction. 外部命令之能读取 Revit® 模型。在该模式下，如何外部命令试图创建 Transaction 或者修改 Revit® 模型，Revit® 将抛出异常。

需要注意的是，上述三种 TransactionMode 都只在活动文档范围内有效。

2) Journaling 属性

Journaling 属性是一个可选属性，用户可以自行选择使用或者不使用。该属性用于控制 Revit Journal 文件在执行外部命令过程中的行为。其格式参见代码片段 2-13。

代码片段 2-13：外部命令中 Excute 函数的 Journaling 属性

```
[Autodesk. Revit. Attributes. Journaling (Autodesk. Revit. Attributes. JournalingMode.
    NoCommandData)]
public class Class1: IExternalCommand
{
}
```

下面为当前 Revit® API 支持的 Journaling 属性模式：

① JournalMode. NoCommandData：使用该模式，Revit® 将不会把 ExternalCommandData. JournalData 的内容写到 Revit® Journal 中。

② JournalMode. UsingCommandData：使用该模式，Revit® 使用 IDictionary<String，String>格式把 ExternalCommandData. JournalData 的内容写到 Revit® Journal 中。该模式为 Journaling 属性的默认模式。

2.2 Revit® 应用类和文档类(Application/Document)

Revit® 与 Microsoft Office 一样，使用多文档模型。在 Revit® API 的类结构中，应用类和文档类是处于最顶层的。它们主要分为 Application，UIApplication，Document 和 UIDocument 这几大类。

(1) 一个 Revit® 应用对象对应一个独立的 Revit® 会话。用户可以通过这个对象访问 Revit® 文档、选项以及其他应用范围的数据和设置。

① Autodesk. Revit. UI. UIApplication：提供访问 UI 级别接口的应用类，包括访问用

户界面 RibbonPanels、获得用户界面的活动文档等能力。

② Autodesk. Revit. ApplicationServices. Application：提供访问其他应用级别内容接口的应用类。

(2) 一个 Revit® 文档对象对应一个独立的 Revit® 工程文件。Revit® 可以同时有多个工程打开，每个工程可以同时有多个视图。

① Autodesk. Revit. UI. UIDocument：提供访问 UI 级别接口的文档类，比如提供 Selection 来表示用户界面的选择集。

② Autodesk. Revit. DB. Document：提供访问其他文档基本内容的文档类。

(3) 如果有多文档打开，那么处于最顶层或者活动的视图所在的工程或者文档就是活动文档。

本章主要介绍 Revit® 应用类和文档类的功能，以及文件管理、设置、单位等方面。如果想了解 Revit® 元素类方面的信息，请参见第 3 章。

2.2.1　Revit® 应用类

Revit® 应用类主要分为 UIApplication 和 Application 两大类。在这一节中将分别介绍 Application 和 UIApplication。

Revit® API 中还有两类 Application，分别为 UIControlledApplication 和 ControlledApplication。这两类 Application 是对 UIApplication 和 Application 的进一步封装，用户无法通过这两个应用类访问 Revit® 文档。用户通过这两类 Application 只可以实现 UI 定制和一些事件处理。

1) Application

Application 类表示一个 Revit® 应用，提供对文档、选项以及其他应用范围的数据的访问和设置。

(1) 获取 Application 对象

用户可以通过 IExternalCommand 的 Excute 函数中的 ExternalCommandData 来获取 Application 对象，参见代码片段 2-14。

代码片段 2-14：获取 Application 对象

```
Autodesk.Revit.ApplicationServices.Application app
= commandData.Application.Application;
```

(2) Revit® 版本及产品信息

Revit® Application 对象里面包括了以下的 Revit® 版本及产品信息。它们可以为插件提供 Revit® 的发布信息。

- VersionBuild
- VersionNumber
- VersionName
- Product
- Language

代码片段 2-15 演示如何使用这些属性。

代码片段 2-15：Revit® 版本及产品信息

```
public void GetVersionInfo(Autodesk.Revit.ApplicationServices.Application app)
{
  if (app.VersionNumber = = "2014")
  {
TaskDialog.Show("Supported version",
"This application supported in this version.");
  }
  else
  {
TaskDialog dialog = new TaskDialog("Unsupported version.");
      dialog.MainIcon = TaskDialogIcon.TaskDialogIconWarning;
      dialog.MainInstruction = "This application is only supported in Revit 2014.";
      dialog.Show( );
  }
}
```

（3）共享参数管理

Revit® 提供了共享参数机制。Application 类中的 OpenSharedParameterFile()方法可以用来访问共享参数文件。Application 类中的 SharedParametersFilename 属性提供了共享参数文件的名称。更多关于共享参数的介绍，请参见本书“3.1 元素基础”。

2）UIApplication

UIApplication 表示一个活动的 Revit® 会话，提供了对 UI 定制、事件以及活动文档的访问。

（1）获取 UIApplication 对象

用户可以通过 IExternalCommand 的 Excute 函数中的 ExternalCommandData 来获取 UIApplication 对象，参见代码片段 2-16。

代码片段 2-16：获取 UIApplication 对象

```
Autodesk.Revit.UI.UIpplication uiApp = commandData.Application;
```

（2）插件管理

UIApplication 类提供了 ActiveAddInId 属性来获取当前活跃的外部命令或者外部应用的 ID。同时 UIApplication 类也提供了 LoadedApplications 属性来返回一个成功加载插件的列表。

（3）UI 扩展

UIApplication 提供了创建自定义 Ribbon 面板和控件的方法。想了解更多 UI 扩展的信息，请参见本书第 10 章功能区扩展(Ribbon UI)。

2.2.2 Revit® 文档类

Revit® 文档类主要分为 UIDocument 和 Document 两大类。在这一节中将分别介绍这两种文档类。

1）Document

Document 类用于表示打开的 Autodesk® Revit® 工程。Revit® 可以同时有多个工程打开，即同时拥有多个 Document 对象。Document 对象用于存储 Revit® 的元素、管理视图和数据。Document 类主要提供获取管理元素、文件管理等功能。

（1）位置和场地

每个工程有唯一的位置属性（对应 UI 为“地点”，对应 API 为 SiteLocation）来标识这个工程在地球上的地理位置。同时，一个工程可以有很多的场地属性（对应 API 为 ProjectLocation），每个场地属性可以是一个相对于地理位置的位移，也可以是一个相对于地理位置的旋转。

Document 类提供了 SiteLocation 属性来取到项目地理位置信息。在 Revit® 中每个工程应该有一个场地信息来表示这个工程位于世界的哪个位置。表 2-3 列出了 SiteLocation 类的属性。

表 2-3　　位置和场地

属性	说明
Elevation	当前位置的海拔
Latitude	当前位置所处纬度
Longitude	当前位置所处经度
PlaceName	当前位置的地名
TimeZone	当前位置所在时区
WeatherStationName	当前位置使用的气象站名

Document 类提供了 ProjectLoactions 属性来取到项目场地集合（ProjectLoactionSet）。用户可以遍历这个集合来取到当前项目的所有场地信息（ProjectLocation）。

一个 ProjectLocation 对象用来表示项目中的一个命名场地，即一个模型实体在项目中的位置，它包括一个 ProjectPosition（表示相对于地理位置的偏移或旋转）和一个 SiteLocation（这个场地对应于的地理信息对象）。只有 Revit® Architecture 工程可以有命名场地。一个 Revit® Architecture 项目默认有一个命名场地，如图 2-8 所示的是 UI 界面上的默认命名场地——室内灯。

用户可以通过调用 ProjectLoacation. Duplicate (String)来创建一个新的 ProjectLocation 对象。

（2）视图管理

一个 Revit® 工程文档可以有多个视图。通过 Document 类的 ActiveView 属性可以取到当前活动的视图对象。如果想获取全部的视图对象，可以通过过滤器来实现。更多视图的详细介绍请参见本书第 8 章视图（Views）。



图 2-8　UI 界面上的默认命名场地

（3）获取元素

Revit® 文档对象存储了对应 Revit® 工程里的全部元素。用户可以通过文档对象提供的以下方

法来获取元素：

- 通过 ElementId 获取对应的元素；
- 通过引用获取对应的元素；
- 通过 UniqueId 获取对应的元素。

如果用户想获取多个同类别的元素，可以通过过滤器来实现。更多信息请参见本书 3.3 节。

2）UIDocument

UIDocument 对象主要包括了跟 UI 相关的文档信息和操作，比如 ActiveView 和 ShowElement。它主要提供以下功能：

（1）获取元素

UIDocument 类提供了 Selection 属性，用户可以通过这个属性取到被选中的元素。这个属性返回一个被选中的对象的集合。这个属性统一提供了一些 UI 交互方法可以让用户在插件执行过程选择 Revit® 模型。请参见本书第 2 章 2.4.3 节。

（2）显示元素

UIDocument 类提供了一系列的 ShowElements()方法缩放视图来显示适当的元素。

（3）视图管理

UIDocument 类提供的视图管理主要是与 ActiveDocument 相关的。

- UIDocument. ActiveView 属性可以返回或设置当前的文档的活动视图；
- UIDocument. RefreshActiveView()方法可以用来刷新当前文档的活动视图；
- UIDocument. RequestViewChange()可以发出一个改变当前文档的视图的异步请求。

UIDocument 类也可以取到当前文档中打开的全部视图的列表。GetOpenUIViews()方法返回一个 UIViews(表示 Revit® 界面视图框中的信息)的列表。

2.2.3 Revit® 文档和文件管理

Revit® API 提供了一系列的方法和属性来实现对文档和文件的管理。

1）获取文档对象

Revit® Application 类中包含全部文档。就像前面提到的一样，用户可以在一个 Revit® 会话中打开多个文档，如表 2-4 所示。

表 2-4　　获取文档对象

属性	说明
UIApplication. ActiveUIDocument	属性取到当前的活动文档
Application. Documents	可以取到当前会话中所有打开的文档，包括活动文档

参见代码片段 2-17。

代码片段 2-17：获取文档对象

```
Autodesk.Revit.UI.UIDocument activeDoc = commandData.Application.ActiveUIDocument;
Autodesk. Revit. DB. DocumentSet documents = commandData. Application. Application.
Documents;
```

2）获取文档的文件信息

Revit® Document 类提供了 PathName 和 Title 两个属性来得到文件信息，如表 2-5 所示。

表 2-5　　获取文档的文件信息

属性	说　明
PathName	返回文档的全路径。如果当前文档是新创建的并且没有保存过，那么该属性返回值为空
Title	返回文档的工程名称。根据系统设置的不同，该属性会返回带后缀名的工程名称和不带后缀名的工程名称

3）创建文档

用户可以使用表 2-6 列出的 Application 类的方法来创建 Revit® 文档类对象。

表 2-6　　创建文档

方法	说明
DocumentNewFamilyDocument(string templateFileName)	新建一个族文档
DocumentNewProjectDocument(string templateFileName)	新建一个工程文档
DocumentNewProjectTemplateDocument(string templateFilename)	新建一个工程模板文档

每个方法都需要一个模板文件作为参数。新创建的文档根据模板文件的不同而不同。

4）打开文档

Revit® Application 类提供了表 2-7 列出的方法来打开 Revit® 文档。

表 2-7　　打开文档

属性	说　明
Document OpenDocumentFile (string fileName)	当用户指定了一个文件的路径，Revit® 会打开文件并创建文档实例。使用这个方法来打开其他计算机上的文件时，需要传入一个符合通用命名转换规范的文件名。文件类型可以是扩展名为. rvt 的工程文件，也可以是扩展名为. rfa 的族文件或者扩展名为. rte 的模板文件
Document OpenDocumentFile (ModelPath modelPath, OpenOptions openOptions)	使用 ModelPath 类型来管理文件路径，使用 OpenOption 对象来管理打开文件的方式。ModelPath 参数是用来表现共享文件（云文件）的一个集合，包括服务器路径等信息。OpenOption 参数可以控制诸如打开中心文件时是否要从中心分离时等选项。这个方法更多的用在协同工作环境下

5）保存和关闭文档

Revit® Document 类提供了表 2-8 列出的方法来保存文档。

表 2-8　　保存文档

方法	说　明
Save()	保存文档。如果文档从未被保存过，Revit 将调用 SaveAs()来代替 Save()
Save(SaveOptions options)	保存文档。SaveOptions 参数用于指定是否强制删除原文档在磁盘上的所有无效数据
SaveAs(string filename)	将文件另存为 filename 参数指定的文件。如果已经有同名文件存在，该函数将抛出异常
SaveAs (string filename, SaveAsOptions options)	将文件另存为 filename 参数指定的文件。SaveAsOptions 参数用于指定如果在文件名已经存在的情况下，是否重命名或者是否覆盖已有文件。SaveAsOptions 也可以用于设定其他 Save 相关选项，比如是否强制删除原文档在磁盘上的无效，以及协同工作（workSharing）相关的选项
SaveAs (ModelPath path, SaveAsOptions options)	将文件另存为 ModelPath 参数指定的文件位置和文件名

Revit® Document 类提供了表 2-9 列出的方法来关闭文档。

表 2-9　　关闭文档

方法	说　明
Close()	使用该方法来关闭文档时，如果文档被修改过，那么 Revit®将在文档关闭前弹出对话框询问用户是否需要保存文件。如果文档路径名称没有指定或者目标文件已经存在并且是只读的，该方法将抛出异常
Close(bool saveModified)	使用该方法，用户可以指定是否在关闭前保存文档

需要注意的是，Close()方法不能用于关闭 Active Document。这两个方法只能用于关闭非活动文档。

Revit® UIDocument 类提供了表 2-10 列出的两个方法来保存关闭文档。

表 2-10　　保存关闭文档

方法	说明
SaveAndClose()	使用该方法来保存关闭文档时，如果文档的路径名称没有指定，Revit®将弹出“另存为”对话框让用户设置文件的名称和保存位置
SaveAs()	使用该方法来另存文档时，Revit®将弹出“另存为”对话框让用户设置文件的名称和保存位置

2.2.4　类别(Category)

在 Revit® API 中，所有基类为 Element 类的元素都拥有一个 Category 属性用于表示这个元素属于的类别或者子类别。Document. Settings 提供了 Categories 属性，用户可以通过这个属性取到一个包含了文档全部 Category 对象的图。

1) Categories 分类

在 Revit®中，Categories 可以分为下面几大类别：

- 模型类别 Model Categories：包括梁、柱、门、窗、墙等。
- 标注类别 Annotation Categories：包括尺寸、网格、标高、文本标注等。

2) Categories 遵循的准则

Revit®中，Categories 遵循以下准则：

- 基本准则：每个族对象属于一个类别；非族对象，比如材质、视图等，不属于任何类别。
- ProjectInfo 类是个例外，它属于 Project Information 类别。
- 一个元素和它的类型通常都属于同一个类别。例如，一个基础墙和它的类型 Generic-8″都属于 Wall 类别。
- 同一个元素类型可以属于不同的类别。例如，SpotDimensionType 既属于 Spot Elevations 类别，也属于 Spot Coordinates 类别。
- 不同的元素可以属于同一个类别(由于相似，或者建筑学上的原因)。例如，ModelLine 和 DetailLine 都属于 Lines 类别。

3) 访问文档中的类别

用户可以通过下面两种方式访问文档中的类别：

- 从 Document. Settings. Categories 属性取到全部的类别。
- 使用 BuiltInCategory 枚举类型从类别图中快速取到指定的类别。

代码片段 2-18 演示如何从文档类属性中取到全部的类别。

代码片段 2-18：从 Setting 中取到当前文档的 Categories

```
//从当前文档对象中取到 Setting 对象
Settings documentSettings =  document.Settings;
String prompt =  "Number of all categories in current Revit document: " +  documentSettings.
Size+ "\n";

//用 BuiltInCategory 枚举值取到一个对应的 Floor Category，打印其名字
Category floorCategory =  documentSettings.get_Item(BuiltInCategory.OST_Floors);
prompt + = "Get floor category and show the name: ";
prompt + =  floorCategory.Name;

TaskDialog.Show("Revit", prompt);
```

4）Category 的用途

Category 有以下三类用途：

- Category 可以用来作为元素分类的方法。例如，同一类别的元素可以被列入同一个明细表中。
- 元素根据类别不同，参数也不相同。
- 类别也可以用来控制可见性和图形外观，如图 2-9 所示。

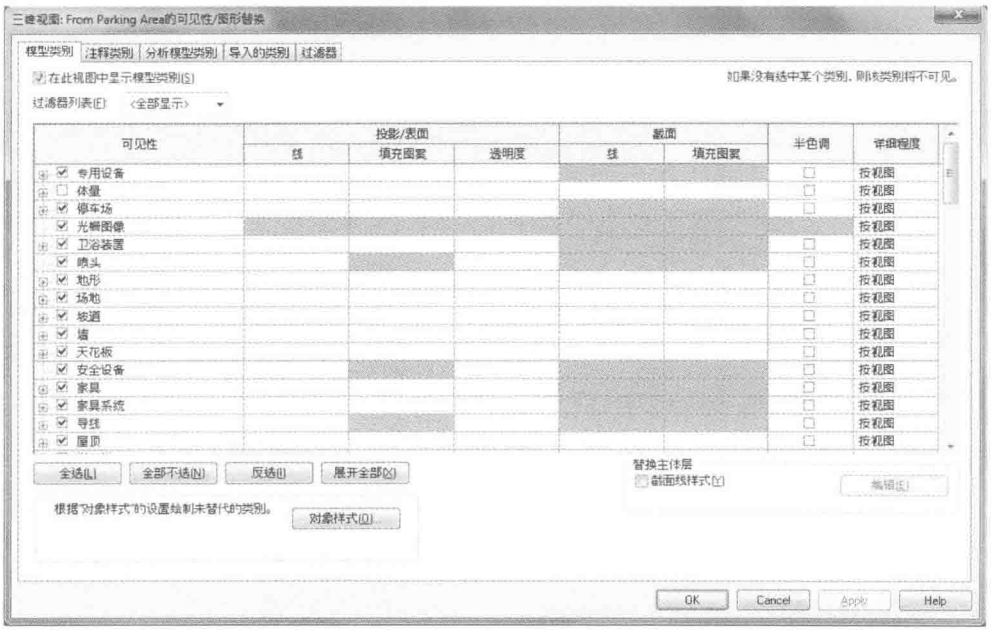


图 2-9　控制可见性和图形外观

5) Category Id

一个元素的类别由它的 Category Id 所决定。

- Category Id 也是用 ElementId 来表现的。
- Revit® 中大部分的类别都是内建的，它们的 Id 值是作为常量保存在 ElementIds 中的。
- 每一个内建的类别 Id 在 BuiltInCategory 枚举中都有对应的值。它们可以被转换成对应的 BuiltInCategory 枚举类型。

代码片段 2-19 演示如何使用 Category Id。

代码片段 2-19：使用 Category Id

```
Element selectedElement = null;
foreach (Element e in document.Selection.Elements)
{
        selectedElement = e;
        break;
}
//取到当前元素的类别
Category category = selectedElement.Category;
BuiltInCategory enumCategory = (BuiltInCategory)category.Id.Value;
```

2.2.5 打印

打印是 Revit® 很常用的功能之一，因此 Revit® 文档类提供了一系列的方法来通过 API 实现打印功能。用户可以通过 API 实现自己的插件，从而将打印进行批量化的处理。

代码片段 2-20 演示如何使用 Revit® API 实现打印功能。

代码片段 2-20：打印当前文档中的可打印视图

```
public Autodesk.Revit.UI.Result Execute(ExternalCommandData commandData, ref string
    message, Autodesk.Revit.DB.ElementSet elements)
  {
      Document doc = commandData.Application.ActiveUIDocument.Document;
      FilteredElementCollector collector = new FilteredElementCollector(doc).OfClass
      (typeof(ViewPlan));
      IList< Element> viewElems = collector.ToElements( );
      ViewSet printableViews = new ViewSet( );

//找出全部可打印视图
      foreach (View view in viewElems)
      {
        if (! view.IsTemplate && view.CanBePrinted)
        {
            printableViews.Insert(view);
        }
      }
      PrintManager pm = doc.PrintManager;
```

```
        pm.PrintRange = PrintRange.Select;
        pm.SelectNewPrintDriver(@ "\\server\printer01");
        pm.Apply( );

//打印全部可打印视图
        doc.Print(printableViews);
        return IAsyncResult.Succeeded;
    }
```

2.3　事务处理（Transaction）

2.3.1　事务简介

事务是将一系列修改 Revit®模型的操作提交至文档的这样一种对象。任何一个对文档修改的操作都需要包含在一个打开的属于该文档的事务中，否则就会有异常抛出。任何修改都要在事务提交后才写入文档。在一个事务中进行的所有修改都能被撤销。在任何时候，一个文档当前只能有一个打开的事务，但在一个事务中可以有一个或多个修改模型的操作。

在 API 中和事务有关的主要有三个类：

- Transaction
- SubTransaction
- TransactionGroup

本章将会深入介绍这三个类。其中，只有 Transaction 类才是修改文档必须使用的。其他两个主要用来更好地对事务进行管理。

注意：

① 用于 IExternalCommand 的 TransactionMode 属性会影响该命令在被执行时 Revit®对事务的处理。

② 如果一个事务从其他线程或在模态对话框之外启动会导致异常发生。事务只能从 API 支持的环境中启动，比如外部命令、事件等。

2.3.2　事务类

所有三个事务相关的类有相同的方法，如表 2-11 所示。

表 2-11　　通用事务方法

方法	描　述
Start	启动事务
Commit	结束事务并提交所有的修改到文档中
RollBack	结束事务并撤销对文档所做的所有的修改
GetStatus	返回当前事务状态

除了 GetStatus 函数会返回当前事务状态外，Start，Commit 和 RollBack 函数也会返回 TransactionStatus 表示当前操作是否成功。TransactionStatus 值包含表 2-12 列出的值。

表 2-12　　TransactionStatus 值

值	描述
Uninitialized	表示当前是否已经被初始化
Started	事务已经成功启动(Start 已经被调用)
RolledBack	事务被成功撤销(Rollback 已经被调用)
Committed	事务已经被成功提交(Commit 已经被调用)
Pending	事务试图被提交或撤销。但由于出错,处理还未结束,还在等待用户响应(在模态对话框中)。一旦错误处理完,状态会自动更新为已提交或撤销(Committed 或 RolledBack)

1) Transaction 类

要修改 Revit® 文档中的模型就需要一个 Transaction。在同一时间,只有一个 Transaction 可以被打开,不允许嵌套。每个 Transaction 需要一个名字,当这个 Transaction 被成功提交后,这个名字会被显示在 Undo 菜单里。代码片段 2-21 演示了如何使用 Transaction。

代码片段 2-21:使用事务创建元素

```
public void CreatingSketch(UIApplication uiApplication)
{
    Document document = uiApplication.ActiveUIDocument.Document;
    ApplicationServices.Application application = uiApplication.Application;

    //创建一些几何线,这些线是临时的,所以不需要放在事务中
    XYZ Point1 = XYZ.Zero;
    XYZ Point2 = new XYZ(10, 0, 0);
    XYZ Point3 = new XYZ(10, 10, 0);
    XYZ Point4 = new XYZ(0, 10, 0);

    Line geomLine1 = Line.CreateBound(Point1, Point2);
    Line geomLine2 = Line.CreateBound(Point4, Point3);
    Line geomLine3 = Line.CreateBound(Point1, Point4);

    //这个几何平面也是临时的,不需要事务
    XYZ origin = XYZ.Zero;
    XYZ normal = new XYZ(0, 0, 1);
    Plane geomPlane = application.Create.NewPlane(normal, origin);

    //为了创建 SketchPlane,我们需要一个事务,因为这个会修改 Revit® 文档模型

    //任何 Transaction 要放在"using"中创建
    //来保证它被正确的结束,而不会影响到其他地方
    using (Transaction transaction = new Transaction(document))
    {
      if (transaction.Start("Create model curves") == TransactionStatus.Started)
      {
          //在当前文档中创建一个 SketchPlane
          SketchPlane sketch = SketchPlane.Create(document, geomPlane);
```

```
            //使用 SketchPlane 和几何线来创建一个 ModelLine
              ModelLine line1 =  document.Create.NewModelCurve(geomLine1, sketch) as
              ModelLine;
              ModelLine line2 =  document.Create.NewModelCurve(geomLine2, sketch) as
              ModelLine;
              ModelLine line3 =  document.Create.NewModelCurve(geomLine3, sketch) as
              ModelLine;

            //询问用户这个修改是否要提交
            TaskDialog taskDialog =  new TaskDialog("Revit");
            taskDialog.MainContent =  "Click either [OK] to Commit, or [Cancel] to Roll back
            the transaction.";
                   TaskDialogCommonButtons  buttons  =  TaskDialogCommonButtons.Ok  |
                   TaskDialogCommonButtons.Cancel;
            taskDialog.CommonButtons =  buttons;

            if (TaskDialogResult.Ok = =  taskDialog.Show( ))
            {
              //由于种种原因,如果修改或创建的模型不正确,
              //这个 Transaction 可能不会被正确提交
              //如果一个 Transaction 失败了或被用户取消了,
              //那么返回的状态就是 RolledBack,而不是 Committed。
              if (TransactionStatus.Committed ! =  transaction.Commit( ))
              {
                  TaskDialog.Show("Failure", "Transaction could not be committed");
              }
            }
            else
            {
              transaction.RollBack( );
            }
        }
    }
}
```

2) SubTransaction 类

一个 SubTransaction 可以用来提交一组操作。子事务是可选项,不是必须的。它们是用来将一个复杂的任务分解成许多小任务来提交。SubTransaction 必须在一个打开的 Transaction 中创建,在 Transaction 关闭(提交或撤销)前关闭(提交或撤销)。与事务不同的是,子事务可以嵌套,但嵌套内的子事务必须在嵌套外的子事务关闭前关闭。子事务没有名字,因为它不会显示在 Undo 菜单里。

3) TransactionGroup 类

TransactionGroup 用来将几个独立的事务分组,这样一个组可以同时处理许多事务。当一个 TransactionGroup 撤销时,所有在这个组里的事务,哪怕已经成功提交的,都会被撤销。如果一个 TransactionGroup 被提交,这个组里的事务保持原来状态(提交或撤销)。如果一个 TransactionGroup 被吸收(Assimilate),那么这个组里的所有事务将被合并成一个成功提交的事务,并使用这个组的名字显示在 Undo 菜单里。

一个 TransactionGroup 只能在没有事务打开的情况下启动，也必须在所有事务已经关闭的情况下关闭。TransactionGroup 可以被嵌套，但任何嵌套内的事务组必须在嵌套外的事务组关闭前关闭。事务组主要用来管理事务，不是必须使用的。

代码片段 2-22 展示了如何使用 TransactionGroup 的 Assimilate 方法来组合两个独立的事务。运行结果将在 Undo 菜单里显示一个事务。

代码片段 2-22：使用 Assimilate 方法

```
public void CompoundOperation(Autodesk.Revit.DB.Document document)
{
    //所有 TransactionGroup 要用"using"来创建来保证它的正确结束
     using (TransactionGroup transGroup = new TransactionGroup(document, "Level and
     Grid"))
    {
      if (transGroup.Start( ) = = TransactionStatus.Started)
      {
          //我们打算调用两个函数，每个都有一个独立的事务
          //我们打算这个组合操作要么成功，要么失败
          //只要其中有一个失败，我们就撤销所有操作

          if (CreateLevel(document, 25.0) && CreateGrid(document, new XYZ(0, 0, 0), new
          XYZ(10, 0, 0)))
          {
            // Assimilate 函数会将这两个事务合并成一个，并只显示 TransactionGroup 的名字
            //在 Undo 菜单里
            transGroup.Assimilate( );
          }
          else
          {
            //如果有一个操作失败了，我们撤销在这个事务组里的所有操作
            transGroup.RollBack( );
          }
      }
    }
}

public bool CreateLevel(Autodesk.Revit.DB.Document document, double elevation)
{
    using (Transaction transaction = new Transaction(document, "Creating Level"))
    {
      //必须启动事务来修改文档
      if (TransactionStatus.Started = = transaction.Start( ))
      {
          if (null ! = document.Create.NewLevel(elevation))
          {
            return (TransactionStatus.Committed = = transaction.Commit( ));
          }
          //如果不能创建层，撤销这个事务
          transaction.RollBack( );
      }
```

```
    }
    return false;
}

public bool CreateGrid(Autodesk.Revit.DB.Document document, XYZ p1, XYZ p2)
{
    using (Transaction transaction =  new Transaction(document, "Creating Grid"))
    {
      if (TransactionStatus.Started = =  transaction.Start( ))
      {
          Line gridLine =  Line.CreateBound(p1, p2);

          if ((null ! =  gridLine) && (null ! =  document.Create.NewGrid(gridLine)))
          {
            if (TransactionStatus.Committed = =  transaction.Commit( ))
            {
                return true;
            }
          }
          //如果不能创建网格,撤销这个事务
          transaction.RollBack( );
      }
    }
    return false;
}
```

2.3.3　在事件中使用事务

1）在事件中修改文档

事件不会自动启动事务。要在事件响应中修改文档，必须显式地启动一个事务，在该事务中修改文档。如果是在事件中打开事务，则事务必须在该事件响应结束前被关闭（提交或撤销），否则在该事务中所做的所有的修改都会被丢弃。

需要注意的是，在一些事件中修改当前文档是被禁止的（如 DocumentClosing 事件）。如果试图在这些事件响应中修改文档，异常会被抛出。具体可以查询本书第 9 章来确定哪些事件是不允许修改文档的。

2）事务调用触发的事件

DocumentChanged 事件在每个事务被提交、撤销或重做的时候被触发。这个是只读事件，允许用户获取哪些元素被添加、修改或删除了，则可以用它来使外部数据和 Revit® 文档数据保持同步。

2.4　实战示例

实战小节将从以下四个方面来帮助用户快速使用 Revit® API：

① 实战 1："Hello Revit"。介绍如何用 Revit® API 创建一个插件。

② 实战 2：将"Hello Revit"添加到菜单。介绍如何自定义按钮，设计按钮图标。

③ 实战 3：从 Revit® 中选取元素。介绍如何选取 Revit® 文档中的元素。

④ 实战 4：从 Revit® 中过滤元素。介绍如何通过过滤器来获取 Revit® 文档中的元素。

2.4.1 实战 1："Hello Revit"

按以下步骤创建一个最简单的项目，该插件作为一个 Revit® 外部命令，执行时弹出一个提示框显示"Hello Revit!"。

1）新建项目

打开 Visual Studio 2010，单击菜单框"File"→"New"→"Project…"，打开"New Project"对话框，如图 2-10 所示。

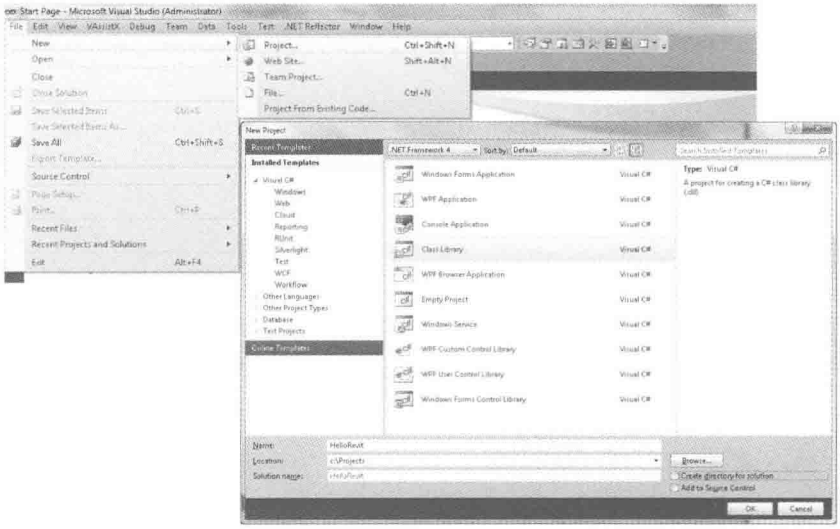


图 2-10 "新建项目"对话框

在该对话框中，在"Project type"中选择"Visual C#"→"Windows"，在"Templates"中选择"Class Library"，在"Name"栏中输入项目名称"HelloRevit"，点击"OK"完成。

2）添加引用

选择"Solution Explorer"的"References"节点（如果"Solution Explorer"窗口没有打开，可以通过点击菜单"View"→"Solution Explorer"打开）。右击选择"Add Reference…"，打开"Add Reference"对话框，见图 2-11。

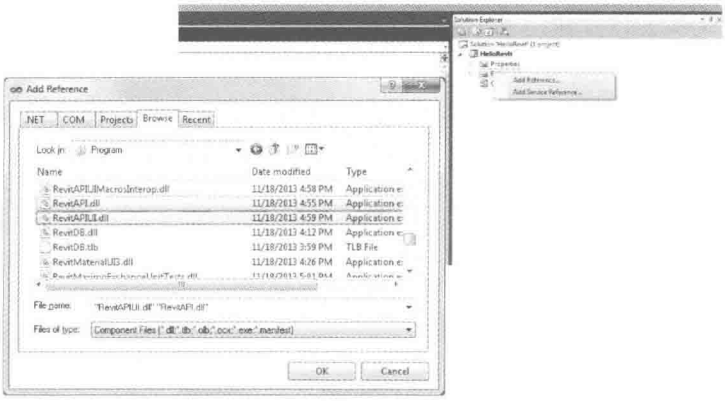


图 2-11 "添加引用"对话框

在该对话框中，选择“Brower”选项卡，进入 Revit® 安装目录（例如：C:\Program Files\RevitArchitecture 2015\Program），选择“RevitAPI. dll”以及“RevitAPIUI. dll”两个动态链接库。点击“OK”添加到项目引用中。

需要注意的是，在属性中将这两个引用的“CopyLocal”属性值设为“False”，如图 2-12 所示。这样可以避免在编译的时候将这两个文件复制到本地目录。

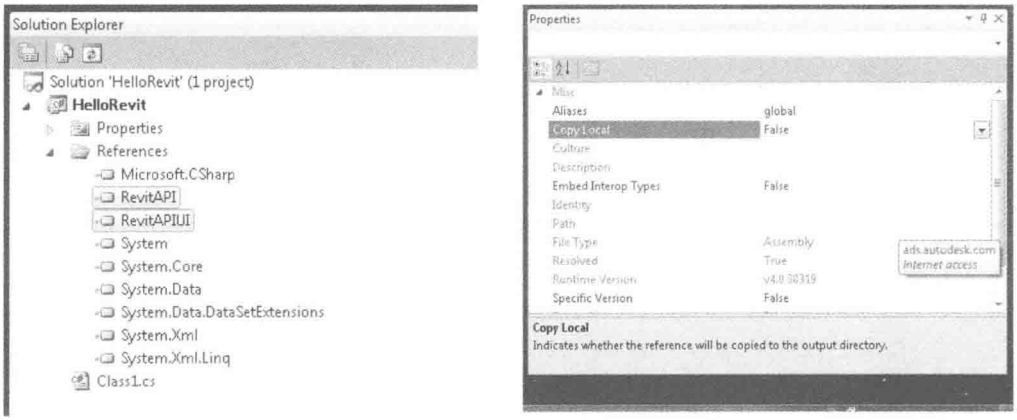


图 2-12　设置属性值

3）代码编写

下面可以开始编写具体代码。请用代码片段 2-23 替换 Visual Studio 默认生成的代码。

代码片段 2-23：HelloRevit

```
using System;

using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
namespace HelloRevit
{
[Autodesk. Revit. Attributes. Transaction (Autodesk. Revit. Attributes. TransactionMode.
Manual)]
  public class Class1: IExternalCommand
  {
      public Autodesk.Revit.UI.Result Execute(ExternalCommandData revit,
          ref string message, ElementSet elements)
      {
        TaskDialog.Show("Revit", "Hello Revit");
        return Autodesk.Revit.UI.Result.Succeeded;
      }
  }
}
```

新建一个文本文件，将文件名改为“HelloRevit. addin”。然后将代码片段 2-24 输入该文件，并保存。

代码片段 2-24：HelloRevit.addin 文件

```
< ? xml version= "1.0" encoding= "utf-8"? >
< RevitAddIns>
< AddIn Type= "Command">
< Assembly> [dll 文件所在路径]\HelloRevit.dll< /Assembly>
< ClientId> 7d4e1893-3a27-4df2-8075-4fa3754537aa< /ClientId>
< FullClassName> HelloRevit.Class1< /FullClassName>
< VendorId> ADSK< /VendorId>
< /AddIn>
< /RevitAddIns>
```

4）执行插件

启动 Revit®，功能区中将出现一个“附加模块”选项卡，见图 2-13。

点击“外部工具”下拉列表中的“HelloRevit”，执行后将弹出一个对话框，见图 2-14。



图 2-13 “附加模块”选项卡

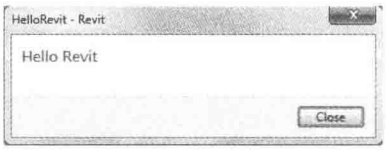


图 2-14 “Hello Revit”对话框

5）调试插件

在编码、运行过程中可能会遇到各种各样的问题，因此需要进行断点调试检查当前变量和对象的状态来定位问题。

- 在“HelloRevit”工程上右击，选择“Properties”，见图 2-15。
- 在打开的对话框左侧面板中选择“Debug”选项，在右侧窗口的“Start Action”中点选“Start external program”，并设置执行文件路径（例如，C:\Program Files\Revit Architecture 2014\Program\Revit.exe），见图 2-16。



图 2-15 右击“Hello Revit”

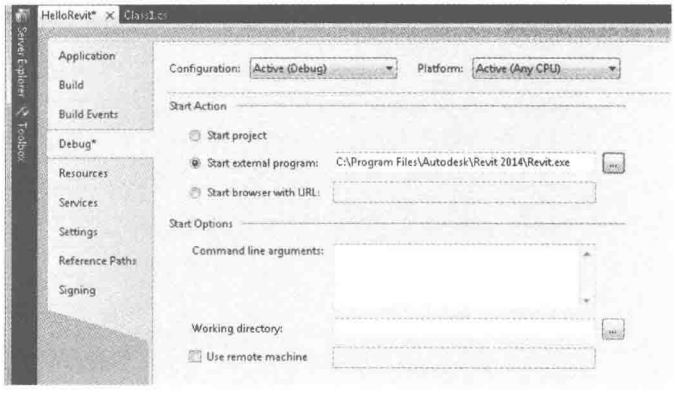


图 2-16 设置执行文件路径

• 编译之后，按 F5 进行调试插件。

另外一种调试方法是，在 Revit® 启动后，在 Visual Studio 的“Debug”菜单下选择“Attach to Process…”(快捷键：Ctrl＋Alt＋P)，见图 2-17。在打开的“Attach to Process”对话框中选择“Revit. exe”，然后单击“Attach”按钮。

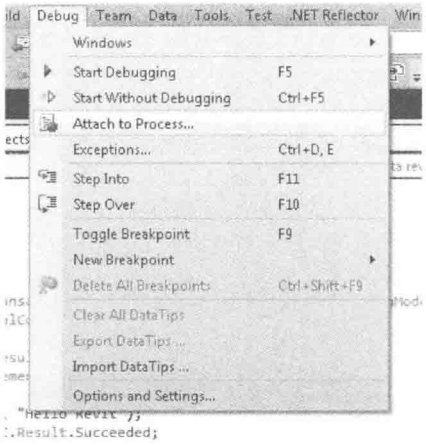


图 2-17　“Debug”菜单

需要注意的是，这种方法不适用于对实现 IExternalApplication 的 OnStartup 调试，因为该方法在 Revit® 启动的时候已经调用了，而“Attach”是在 Revit® 启动后才能操作的。

6) 故障排除

Q：我的插件为什么不能编译？

A：如果在编译实例代码时出现错误，有可能是由于 Revit API 的版本不匹配造成的。请删除老的 Revit API 引用并且添加新版本的 RevitAPI. dll。

Q：为什么我的 Revit® 没有附加模块的选项卡？

A：很多情况下，如果插件应用加载失败，Revit® 在启动的时候会显示一个错误提示。比如，Revit® 在. addin 文件中的指定目录下找不到插件. dll，会显示如图 2-18 所示的对话框。



图 2-18　错误对话框

在错误对话框中，将显示找不到附加模块程序集。

但是，在有些情况下，插件应用加载失败是没有任何提示的。这些情况包括：

• 插件应用所引用的 RevitAPI. dll 与当前 Revit® 版本不匹配。

• Revit® 找不到. addin 文件。

Q：为什么我的插件不工作？

A：如果插件已经出现在附加模块选项卡中，但是却无法工作，这种情况通常是由于插件运行过程中出现了异常，如代码片段 2-25 所示。

代码片段 2-25：插件抛出异常

```
Command: IExternalCommand
{
    public IExternalCommand.Result Execute ( )
    {
      //抛出异常…
    }
}
```

这段代码在 Excute()函数中有异常抛出，Revit®将弹出如图 2-19 所示对话框提示插件执行失败。

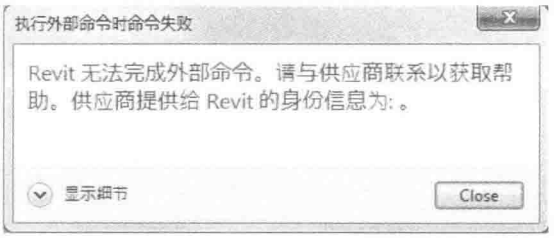


图 2-19 “执行外部命令时命令失败”对话框

2.4.2 实战 2：将“Hello Revit”添加到选项卡

Revit®的功能区 Ribbon 每个一级选项卡下包含多个面板，每个面板又包含多个控件，如按钮、下拉菜单、文字输入框等，见图 2-20。

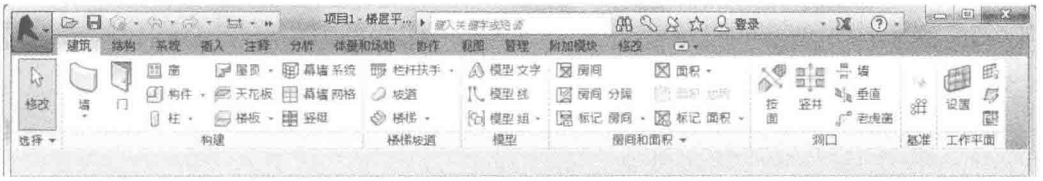


图 2-20 功能区面板

使用上一节所述方法生成的命令都放在 Revit®的 Ribbon 中的“外部工具”下拉列表中，如果开发了多个外部命令，使用起来并不方便。本节将以创建“HelloRevit”按钮为例，介绍如何创建自定义按钮。

要创建自定义按钮，需要实现 IExternalApplication 接口，该类必须同时实现两个接口：OnStartup 和 OnShutdown。前者在 Revit®启动时调用，让用户有机会在 Revit®启动时做一些定制工作；后者在 Revit®关闭时调用，让用户有机会在 Revit®关闭时做一些资源释放或其他工作。

1）代码编写

请将代码片段 2-26 替换 Visual Studio 默认生成的代码。

代码片段 2-26：添加 Ribbon 面板

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows.Media.Imaging;

using Autodesk.Revit.UI;
using Autodesk.Revit.DB;

namespace HelloRevit
{
  public class CsAddpanel: Autodesk.Revit.UI.IExternalApplication
  {
      public Autodesk.Revit.UI.Result OnStartup(UIControlledApplication application)
      {
```

```
//添加一个新的 Ribbon 面板
        RibbonPanel ribbonPanel =  application.CreateRibbonPanel("NewRibbonPanel");

        //在新的 Ribbon 面板上添加一个按钮
        //点击这个按钮,前一个例子"HelloRevit"这个插件将被运行。
        PushButton pushButton =  ribbonPanel.AddItem(new PushButtonData("HelloRevit",
        "HelloRevit", @ "C:\Projects\HelloRevit\HelloRevit.dll", "HelloRevit.Class1")) as
        PushButton;

        //给按钮添加一个图片
        Uri uriImage =  new Uri(@ "C:\Projects\HelloRevit\logo.png");
        BitmapImage largeImage =  new BitmapImage(uriImage);
        pushButton.LargeImage =  largeImage;

        return Result.Succeeded;
      }

      public Result OnShutdown(UIControlledApplication application)
      {
        return Result.Succeeded;
      }
  }
}
```

以上代码主要做了以下工作：

• 通过 UIControlledApplication 的 CreateReibbonPanel 函数创建自定义 Ribbon 面板。

• 创建新的按钮数据 PushButtonData pbd = new PushButtonData("HelloRevit","HelloRevit", @"C:\Projects\HelloRevit\HelloRevit.dll", "HelloRevit.Class1")；其中，PushbuttonData 的第一个参数为内部使用的按钮名称，第二个参数为用户可见的按钮文字，第三个参数为按钮对应的动态链接库(本例即 HelloRevit.dll 的文件路径)，第四个参数为按钮命令实现代码的类名(注意必须包括命名空间路径全名，否则程序无法正确定位命令实现代码)。

• 用刚创建的按钮数据将按钮添加到新创建的 Ribbon 面板上，myPanel.AddItem(pbd)。

• 为创建的按钮添加图片。

2) 注册插件

修改代码后还需要对注册文件进行修改。以文本方式打开之前的"HelloRevit.addin"文件，用代码片段 2-27 替换原来的内容。

代码片段 2-27：HelloRevit.addin 文件

```
< ? xml version= "1.0" encoding= "utf-8"? >
< RevitAddIns>
< AddIn Type= "Application">
< Assembly> C:\Projects\HelloRevit\HelloRevit.dll< /Assembly>
```

```
< AddInId> 6cdba932-c058-4ec1-b038-33ed590c41d3< /AddInId>
< Name> HelloRevit< /Name>
< FullClassName> HelloRevit.CsAddpanel< /FullClassName>
< VendorId> ADSK< /VendorId>
< /AddIn>
< /RevitAddIns>
```

修改完成后，重新启动 Revit®，就可以在“附加模块”选项卡中看到按钮的变化，见图 2-21。

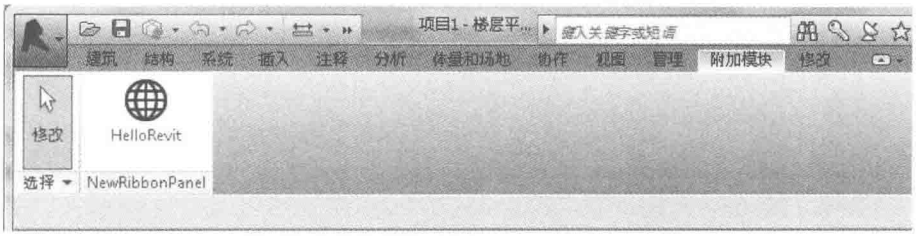


图 2-21 “附加模块”选项卡中出现“HelloRevit”按钮

2.4.3 实战 3：从 Revit® 中选取元素

本节将简要介绍如何通过选中获取文档中的元素。

1）通过选中取到元素的方式

在实现代码前，先介绍一下 Revit® API 的交互方式。Revit® API 提供的交互包括以下几种方式：

（1）先选择元素，再执行外部命令，外部命令可以获取选择元素并做相应处理。

（2）先执行命令，再选择元素或者位置，选择结束后命令继续执行，这种方式的选择又包含几种不同的方式：

① 单选。只能选取一个元素，选择后当前选择操作结束。

② 多选。可以选取多个元素，单击工具条上的“结束”按钮来确认选择操作结束，或者单击“取消”或者按 ESC 键来取消选择。

③ 框选。拉框访方式选取，所有在框内的且符合过滤条件的元素都将被选取。

④ 选点。选择视图中的一个坐标点。

在实际项目中，具体采用何种方式需要根据项目实际情况来决定。因为第二种方式的子类都比较相似（单选、多选、框选），本节只对如何进行单选做介绍。

2）先选择元素后执行命令

可以通过 UIDocument 的 Selection 属性获取到当前视图中选中的元素。下面将用一个简单的例子来说明如何获取当前选择的元素（仍以“HelloRevit”工程为基础）。将 Class1. cs 文件替换为代码片段 2-28。

代码片段 2-28：先选择元素后执行命令

```
using System;
using Autodesk.Revit.UI;
```

```
using Autodesk.Revit.DB;
using Autodesk.Revit.UI.Selection;
namespace HelloRevit
{
  [Autodesk. Revit. Attributes. Transaction (Autodesk. Revit. Attributes. TransactionMode.
Automatic)]
  public class Class1: IExternalCommand
  {
      public Autodesk.Revit.UI.Result Execute(ExternalCommandData revit,
          ref string message, ElementSet elements)
      {

        try
        {
//在执行该插件之前,先选择一些元素。本例中选择了四面墙,一条模型线,一条网格线,一个房间,一
个房间标签。

            //取到当前文档。
            UIDocument uidoc =  revit.Application.ActiveUIDocument;
            //取到当前文档的选择集。
            Selection selection =  uidoc.Selection;
            ElementSet collection =  selection.Elements;

            if (0 = =  collection.Size)
            {
              //如果在执行该例子之前没有选择任何元素,则会弹出提示。
              TaskDialog.Show("Revit", "你没有选任何元素.");
            }
            else
            {
              String info =  "所选元素类型为:";
              foreach (Element elem in collection)
              {
                  info + =  "\n\t" +  elem.GetType( ).ToString( );
              }

              TaskDialog.Show("Revit", info);
            }
        }
        catch (Exception e)
        {
            message =  e.Message;
            return Autodesk.Revit.UI.Result.Failed;
        }

        return Autodesk.Revit.UI.Result.Succeeded;
      }
  }
}
```

先通过 Selection 的 Elements 属性得到所有选中元素的集合，然后遍历该元素集，并把每个元素的名称放到一个字符串中，最后将这些元素的名称显示出来。图 2-22 为运行结果图。

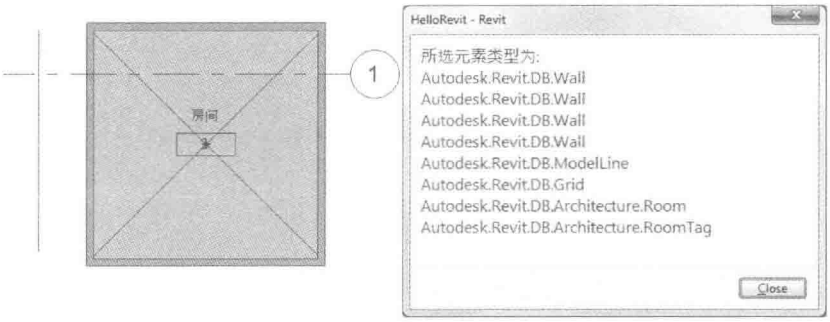


图 2-22　运行结果图

3）在运行外部命令过程中选取元素

Revit® API 支持在运行外部命令的情况下选择元素。请用代码片段 2-29 替换“HelloRevit”工程 Class1. cs 中的代码。

代码片段 2-29：在运行外部命令过程中选取元素

```
using System;
using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI.Selection;
namespace HelloRevit
{
  [Autodesk. Revit. Attributes. Transaction (Autodesk. Revit. Attributes. TransactionMode.
  Automatic)]
  public class Class1: IExternalCommand
  {
      public Autodesk.Revit.UI.Result Execute(ExternalCommandData revit,
          ref string message, ElementSet elements)
      {

        try
        {
            Document document =  revit.Application.ActiveUIDocument.Document;
            //点选指定类型的元素。本例中指定的类型为元素整体。
            Reference pickedElemRef =  this.Selection.PickObject(ObjectType.Element);
            //通过引用取到选中的元素。
            Element elem =  this.Document.GetElement(pickedElemRef.ElementId);
            String info =  "所选元素类型为:";
            info + =  "\n\t" +  elem.GetType( ).ToString( );

            TaskDialog.Show("Revit", info);
        }
        catch (Exception e)
```

```
        {
            message =  e.Message;
            return Autodesk.Revit.UI.Result.Failed;
        }

        return Autodesk.Revit.UI.Result.Succeeded;
      }
  }
}
```

其中 ObjectType. Element 指定选取的是元素整体，Revit® API 还支持选取面(Face)、边(Edge)、点(Point)等。图 2-23 所示是选择一个模型线之后的运行结果。



图 2-23　选择模型线之后的运行结果

2.4.4　实战 4:通过过滤器取到元素

Revit® API 提供 FilteredElement Collector 类来过滤文档的元素，该类提供多种预定义方法对元素进行过滤，这些预定义方法都继承自 ElementFilter。需要注意到是 ElementFilter 分 ElementQuick Filter 和 ElementSlowFilter，详细信息请参见本书“3. 3 元素过滤器”。由于 Revit® 读取元素的方式继承于 ElementQuickFilter 的过滤器，在读取元素信息时只读取有限的元素信息，这些信息不需要把元素的所有信息装载到内存，所以速度更快。而 ElementSlowFilter 则需要更多的信息，会要求元素所有信息读取到内存。因而，在使用过滤器查找元素时，请尽量先使用快速过滤器将大部分不符合条件的元素过滤掉，再使用慢速过滤器处理细节的过滤条件。

下面简单介绍通过类的类型来进行元素过滤。请用代码片段 2-30 替换“HelloRevit”工程 Class1. cs 中的代码。

代码片段 2-30:通过过滤器取到元素

```
using System;

using Autodesk.Revit.UI;
using Autodesk.Revit.DB;
using Autodesk.Revit.UI.Selection;
using System.Collections.Generic;
namespace HelloRevit
{
[Autodesk. Revit. Attributes. Transaction (Autodesk. Revit. Attributes. TransactionMode.
Automatic)]
  public class Class1: IExternalCommand
  {
      public Autodesk.Revit.UI.Result Execute(ExternalCommandData revit,
```

```
            ref string message, ElementSet elements)
        {
            try
            {
                Document document = revit.Application.ActiveUIDocument.Document;

//创建一个类过滤器来过滤出所有的 FamilyInstance 类的元素。
                Element ClassFilter familyInstanceFilter = new ElementClassFilter (typeof
                (FamilyInstance));

//创建一个类别过滤器来过滤出所有的内建类型为 OST_Doors 的元素。
                Element CategoryFilter doorsCategoryfilter = new ElementCategoryFilter
                (BuiltInCategory.OST_Doors);

//创建一个逻辑过滤器来组合前面两个过滤器,实现过滤出所有 Door 元素。
                LogicalAndFilter doorInstancesFilter = new LogicalAndFilter(familyInstanceFilter,
                doorsCategoryfilter);

                FilteredElementCollector collector = new FilteredElementCollector(document);
                ICollection< ElementId> doors = collector.WherePasses(doorInstancesFilter).
                ToElementIds( );

                String prompt = "The ids of the doors in the current document are:";
                foreach (ElementId id in doors)
                {
                  prompt + = "\n\t" + id.IntegerValue;
                }

                TaskDialog.Show("Revit", prompt);

            }
            catch (Exception e)
            {
                message = e.Message;
                return Autodesk.Revit.UI.Result.Failed;
            }
            return Autodesk.Revit.UI.Result.Succeeded;
        }
    }
}
```

该例子中通过定义一个FamilyInstance的类过滤器和一个OST_Doors的内建类别过滤器,并且通过逻辑过滤器将前面两个过滤器组合起来,从而实现过滤出文档中全部的Door元素。运行结果如图2-24所示。

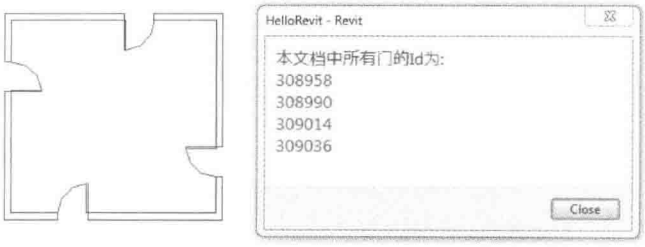


图2-24 元素过滤运行结果

第 3 章　元素(Element)

3.1　元素基础

元素在 Revit® 里面尤其重要，用户能够看见的大多数对象都是元素，比如墙、族、族类型、族实例、标高、轴网、视图等。Revit® 中的大多数类都是继承自元素。

元素是可序列化的，即是可被保存到 RVT 项目文件里。

3.1.1　相关类图

1）族，族实例相关

族、族实例类图见图 3-1，列出了族、族类型和族实例的继承关系。族基本可以对应 Family，族类型对应 FamilySymbol，族实例是 FamilyInstance。它们都是继承自 Element。

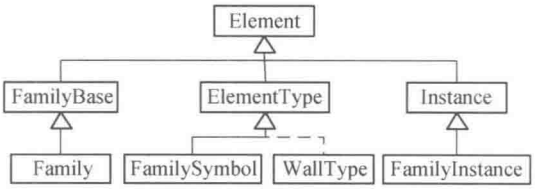


图 3-1　族、族实例类图

2）模型元素相关

模型元素类图见图 3-2，列出的是一些常用的模型元素。类、墙、地板、屋顶、标注、管道、房间等，都是 Element 的子类。

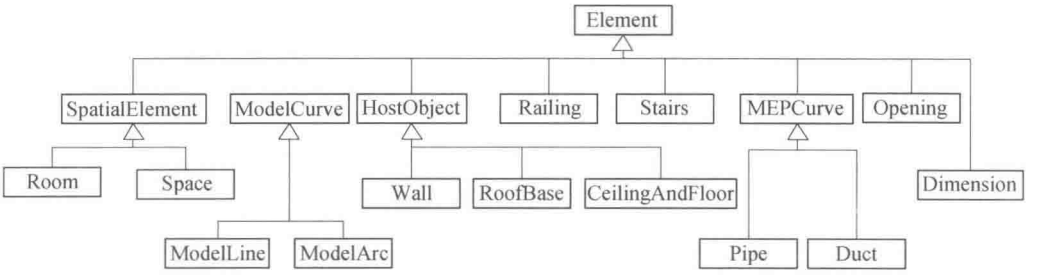


图 3-2　模型元素类图

3）设置相关

设置元素类图见图 3-3，列出的分别是：打印设置，项目信息，太阳和阴影，协同显示设置。



图 3-3　设置元素类图

4）二维元素

二维元素类图见图 3-4 和图 3-5，以上元素都只能在二维视图上面显示。

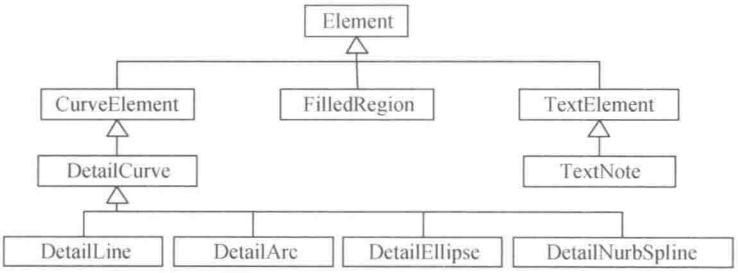


图 3-4　二维元素类图(1)

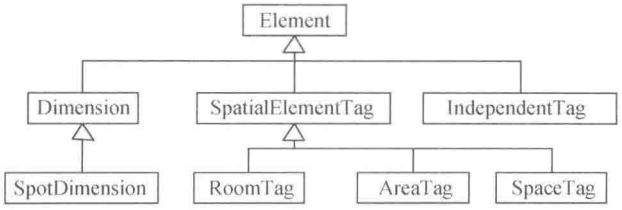


图 3-5　二维元素类图(2)

5) 其他元素

其他元素类图见图 3-6。

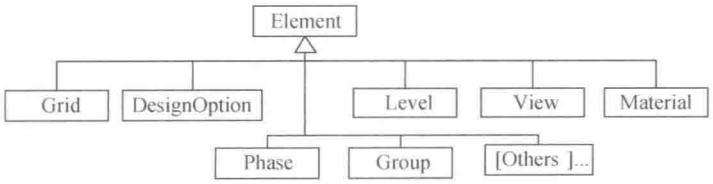


图 3-6　其他元素类图

注意:本书并没有列出所有的类,而是一些重要的基础的类,其他类请使用 ILSpy 或其他工具浏览 RevitAPI. dll,或者参考 RevitAPI. chm 帮助文档。

3.1.2　如何获取元素

1) 通过 ID 获取元素

Revit®中的元素都有一个 ID,我们可以通过 ID 来直接从文档中获取这个元素,代码见代码片段 3-1。

代码片段 3-1:通过 ID 获取元素

```
ElementId levelId =  new ElementId(30);
Element element =  RevitDoc.GetElement(levelId);
Level level =  element as Level;
if(level ! =  null)
{
  //使用 level
}
```

使用上面的方法的缺陷是，用户必须知道这个元素的 ID 才行。

在 Revit® 界面上，可以通过选中这个元素，点击“管理”→“选择项的 ID”来获得，见图 3-7。



图 3-7　查看选择元素的 ID

2）通过过滤器获取元素

过滤器方式是 API 里使用最广的方式，也是推荐使用的方式。

首先创建一个收集器(Collector)：

```
FilteredElementCollector filteredElements =  new FilteredElementCollector(RevitDoc);
```

然后创建一个过滤器：

```
ElementClassFilter classFilter =  new ElementClassFilter(typeof(Wall));
```

接着调用收集器的 WherePasses 函数对元素进行过滤：

```
filteredElements.WherePasses(classFilter);
```

还可以连续调用 WherePasses 函数：

```
filteredElements.WherePasses(classFilter)
.WherePasses(new ElementCategoryFilter(BuiltInCategory.OST_StackedWalls));
```

之后可以遍历收集器，获取用户想要的元素：

```
foreach (Element wall in filteredElements)
{
  //...
}
```

下面的例子展示如何找到文档中所有的外墙，见代码片段 3-2。

代码片段 3-2：过滤所有外墙

```
FilteredElementCollector filteredElements =  new FilteredElementCollector(RevitDoc);
ElementClassFilter classFilter =  new ElementClassFilter(typeof(Wall));
filteredElements =  filteredElements.WherePasses(classFilter);
foreach (Wall wall in filteredElements)
```

```
{
  //获取墙类型"功能"参数,它用来指示墙是否为外墙。
  var functionParameter = wall.WallType.get_Parameter(BuiltInParameter.FUNCTION_PARAM);
  if (functionParameter != null && functionParameter.StorageType == StorageType.
  Integer)
  {
      if (functionParameter.AsInteger( ) == (int)WallFunction.Exterior)
      {
        //使用 wall
      }
  }
}
```

首先创建一个 ElementClassFilter，传入一个 Element 子类类型，然后用 document 创建一个 FilteredElementCollector，接着调用 WherePasses 函数，传入一个创建的 ElementClassFilter，该函数返回一个墙的集合，在遍历的过程中再通过 Wall. WallType 的"功能"属性来判断该墙是不是外墙。

API 提供很多种过滤器，上面的例子是按照类型来过滤元素的，其他常用的还有：

- ElementCategoryFilter，使用类别过滤元素；
- FamilySymbolFilter，使用族类型过滤元素。

过滤器也分快速过滤器、慢速过滤器和逻辑过滤器（详细介绍参见"3. 3 元素过滤器"）。

3）通过选择

使用 UIDocument 的 Selection 属性，用户可以获得被选中的元素集合，如代码片段 3-3 所示。

代码片段 3-3：获取被选元素

```
[Autodesk. Revit. Attributes. Transaction (Autodesk. Revit. Attributes. TransactionMode.
Manual)]
public class MyExternalCommand: Autodesk.Revit.UI.IExternalCommand
{
   public Autodesk. Revit. UI. Result Execute (Autodesk. Revit. UI. ExternalCommandData
   commandData,
      ref string message, ElementSet elements)
  {
      if (commandData.Application.ActiveUIDocument != null)
      {
        foreach (Element selected in commandData.Application.ActiveUIDocument.Selection.
        Elements)
        {
            Wall wall = selected as Wall;
            if(wall != null)
            {
              //使用 wall
            }
```

```
        }
      }
      return Autodesk.Revit.UI.Result.Succeeded;
  }
}
```

3.1.3　获取和修改元素参数

1）获取参数

元素都有参数，可以通过 Element. Parameters 获取所有的参数，然后遍历找到需要的参数。代码片段 3-4 展示如何遍历找到一个元素的“长度”参数。

代码片段 3-4：获取“长度”参数

```
ParameterSet parameters = element.Parameters;
foreach (Parameter parameter in parameters)
{
  if(parameter.Definition.Name == "长度" && parameter.StorageType == StorageType.
  Double)
  {
      double length = parameter.AsDouble( );
      //使用 length
      break;
  }
}
```

也可以通过 Element. get_Parameter(参数)来获取单个参数，这里“参数”有四种选择：string 参数名字、BuiltInParameter 参数枚举、Definition 参数定义和 Guid 参数的 guid。

注意：使用元素名字查找元素的时候，需要注意本地化的问题，因为不同语言的 Revit® 版本的参数名字是不一样的，比如中文版的叫“长度”的参数，英文叫“Length”，同样的程序(比如上面的例子)，在英文里面就会出问题。对此，我们建议尽量使用 BuiltInParameter。

效率上讲，最高的是 BuiltInParameter 参数，其次是 Definition 和 Guid，最慢的是使用参数名字。

代码片段 3-5 展示如何通过 BuiltInParameter 直接获取墙的长度参数。

代码片段 3-5：使用 BuiltInParameter 获取长度

```
Wall wall = null;
Parameter parameterLength = wall.get_Parameter(BuiltInParameter.CURVE_ELEM_LENGTH);
if (parameterLength ! = null && parameterLength.StorageType == StorageType.Double)
{
  double length = parameterLength.AsDouble( );
  //使用 length
}
```

注意：并不是所有的元素都有这个参数，所以用户需要判断获取的参数是不是为空。

2）修改参数

获取到参数之后，进行修改就比较简单了，直接调用 Parameter. Set(参数)这个函数。

代码片段 3-6 展示如何修改墙的“底部偏移”参数：

代码片段 3-6：修改参数

```
Parameter parameterBaseOffset = wall.get_Parameter(BuiltInParameter.WALL_BASE_OFFSET);
if (parameterBaseOffset != null && parameterBaseOffset.StorageType == StorageType.
Double)
{
  if (! parameterBaseOffset.IsReadOnly)
  {
      bool success = parameterBaseOffset.Set(10);
      if (! success)
      {
        //更新错误报告
      }
  }
  else
  {
      //参数是只读的
  }
}
```

注意：

① 设置的值类型一定要和参数的存储类型匹配，如果参数类型是 string，我们调用的是 Set(0. 123)，就会出错。

② 只读参数(即 Parameter. IsReadOnly==true)无法被修改。

3）共享参数和项目参数

我们知道 Revit® 除了有内建的参数(BuiltInParameter)，还允许用户自定义参数，分别是：共享参数和项目参数。对应的界面上的菜单分别是“管理→共享参数”和“管理→项目参数”，见图 3-8。

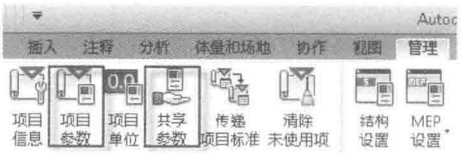


图 3-8 “共享参数”和“项目参数”界面

4）共享参数

共享参数被定义在 Revit® 之外的一个共享参数文件(. txt)中。删掉这个文件，共享参数就会丢失。

在界面上，共享参数好像是一系列的参数，但是实际上只是一些参数的定义，比如参数的名字、类型、分组等。直到把它们绑定到 Category 之后，对应的元素才会出现新的参数。而此时，从某种意义上说，绑定之后的共享参数实际上变成项目参数。

在 API 里面，参数的定义对应的类是 Definition。

代码片段 3-7 展示如何获取 Revit® 里面已有的共享参数。

代码片段 3-7:获取共享参数

```
//打开共享参数文件
DefinitionFile definitionFile = RevitApp.OpenSharedParameterFile( );
//获取参数组的集合
DefinitionGroups groups = definitionFile.Groups;

foreach (DefinitionGroup group in groups)
{
//获取参数组内的参数定义
  foreach (Definition definition in group.Definitions)
  {
      string name = definition.Name;
      ParameterType type = definition.ParameterType;
//对参数定义的其他操作
  }
}
```

如果想要使用 API 创建一个共享参数,可以参考下面的代码片段 3-8。

代码片段 3-8:创建共享参数

```
string sharedParametersFilename = @"C:\shared-parameters.txt";
string groupName = "MyGroup";
string definitionName = "MyDefinition";
ParameterType parameterType = ParameterType.Text;
CategorySet categorySet = new CategorySet( );
Category wallCategory = RevitDoc.Settings.Categories.get_Item(BuiltInCategory.OST_
Walls);
categorySet.Insert(wallCategory);
bool instanceParameter = true;
BuiltInParameterGroup parameterGroup = BuiltInParameterGroup.PG_DATA;

if (! System.IO.File.Exists(sharedParametersFilename))
{
  try
  {
      System.IO.StreamWriter sw = System.IO.File.CreateText(sharedParametersFilename);
      sw.Close( );
  }
  catch (Exception)
  {
      throw new Exception("Can't create shared parameter file: " + sharedParametersFilename);
  }
}
//设置共享参数文件
RevitApp.SharedParametersFilename = sharedParametersFilename;

//打开共享参数文件
DefinitionFile definitionFile = RevitApp.OpenSharedParameterFile( );
if (definitionFile = = null)
```

```
{
  throw new Exception("Can not open shared parameter file!");
}

//获取参数组的集合
DefinitionGroups groups = definitionFile.Groups;

//获取参数组
DefinitionGroup group = groups.get_Item(groupName);
if (null = = group)
{
//如果参数组不存在,则创建一个
  group = groups.Create(groupName);
}
if (null = = group)
  throw new Exception("Failed to get or create group: " + groupName);

//获取参数定义
Definition definition = group.Definitions.get_Item(definitionName);
if (definition = = null)
{
//如果参数定义不存在,则创建一个
  definition = group.Definitions.Create(definitionName, parameterType);
}

//调用不同的函数创建类型参数或者实例参数
ElementBinding binding = null;
if (instanceParameter)
{
  binding = RevitApp.Create.NewInstanceBinding(categorySet);
}
else
{
  binding = RevitApp.Create.NewTypeBinding(categorySet);
}

//把参数定义和类别绑定起来(下面的小节会提到"绑定"),元素的新的参数就创建成功了。
bool insertSuccess = RevitDoc.ParameterBindings.Insert(definition, binding,
parameterGroup);

if (! insertSuccess)
{
  throw new Exception("Failed to bind definition to category");
}
```

5）项目参数

项目参数保存在 Revit® 里，所以删除共享参数或者共享参数文件，都不会对项目参数产生影响，哪怕项目参数是通过共享参数创建的。项目参数在创建的时候，就已经和类别绑定了。也就是说和类别对应的元素都加上了新的参数。

在“项目参数”里看到的参数，实际上是一系列的绑定(Binding)。

注意：通过 API 无法创建项目参数。

代码片段 3-9 展示如何获取类别和参数的绑定。

代码片段 3-9：获取类别和参数的绑定

```
BindingMap map = RevitDoc.ParameterBindings;
DefinitionBindingMapIterator dep = map.ForwardIterator( );
while (dep.MoveNext( ))
{
  Definition definition = dep.Key;
  //获取参数定义的基本信息
  string definitionName = definition.Name;
  ParameterType parameterType = definition.ParameterType;
InstanceBinding instanceBinding = dep.Current as InstanceBinding;
  if (instanceBinding ! = null)
  {
      //获取绑定的类别列表
      CategorySet categorySet = instanceBinding.Categories;
  }
}
```

6) 获取元素的共享参数或项目参数

获取元素的共享参数或项目参数可以通过两种方式：

- 遍历 Element. Parameters 里面的所有参数。
- 通过参数名获取参数 Element. get_Parameter(‘参数名’)。

如何得知这个参数是共享参数还是项目参数？使用 Parameter. IsShared 属性。见代码片段 3-10。

代码片段 3-10：判断共享参数和项目参数

```
Parameter parameter;
InternalDefinition definition = parameter.Definition as InternalDefinition;

bool isSharedParameter = parameter.IsShared;//共享参数

bool isProjectParameter = definition.BuiltInParameter = = BuiltInParameter.INVALID && !
parameter.IsShared;//项目参数
```

3.1.4　重要的属性和方法

元素的几个重要的属性包括：Category(类别)，Location(位置)，LevelId(标高)，GroupId(组)，Id，UniqueId(唯一 Id)，等等。

重要的方法：GetMaterials，GetAnalyticalModel。

1) 元素的位置 Location

Element. Location 属性用来获取元素的位置，Location 可以转型为 LocationPoint 和 LocationCurve，如果该元素的位置是点，则转型为 LocationPoint，如果是直线或者曲线，则

使用 LocationCurve。类图见图 3-9。

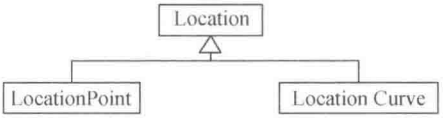


图 3-9 Location 元素类图

2）获取元素的材质

GetMaterials(bool)函数可以获取元素的材质，bool 为 true 的时候，获取的是元素的油漆材质(Painted Material)。

3）获取元素的分析模型

分析模型主要被用来做结构分析，获取分析模型可以使用 Element. GetAnalyticalModel() 方法，然后调用分析模型的 GetCurve，GetCurves 或 GetPoint 来获取分析模型的几何信息，此外，可通过 IsSingleCurve()和 IsSinglePoint()方法来辅助判断需要调用哪个方法。

代码片段 3-11 展示了在什么情况下调用这三个方法。

代码片段 3-11：获取分析模型的几何信息

```
Element element = RevitDoc.GetElement(new ElementId(183554));
if (element = = null) return;
AnalyticalModel analyticalModel = element.GetAnalyticalModel( );
if(analyticalModel.IsSingleCurve( ))
{
  Curve curve = analyticalModel.GetCurve( );
  // work with curve
}
else if(analyticalModel.IsSinglePoint( ))
{
  XYZ p = analyticalModel.GetPoint( );
  // work with point
}
else
{
  IList< Curve> curves = analyticalModel.GetCurves(AnalyticalCurveType.ActiveCurves);
  // work with curves
}
```

3.1.5 元素、族、族类型、族实例

族(Family)，族类型(FamilySymbol)，族实例(FamilyInstance)都是元素(Element)的子类。族可以有多个族类型，一个族类型可以创建多个族实例。

如果用面向对象语言中的概念来讲，族相当于类模板(Class Template)，族类型相当于类(Class)，族实例相当于对象(Object)。虽然它们在 API 中是使用聚合而不是继承的方式组织的，但可以用表 3-1 所示的对应关系来做类比。

表 3-1 元素与 API 中的类的对应关系

元素	API 中对应的类
族	Family
族类型	ElementType (FamilySymbol, WallType, etc.)
族实例	Instance (FamilyInstance, ImportInstance, etc.)

图 3-10 是它们的继承关系。Element 的子类还有很多，这里并没有一一列出来，比较常用的有如 Level 标高，Grid 轴网，View 视图，Material 材质，Dimension 标注，Sketch 草图，宿主元素 HostObject，等等。对这些类的介绍，可以详见其他对应的章节。

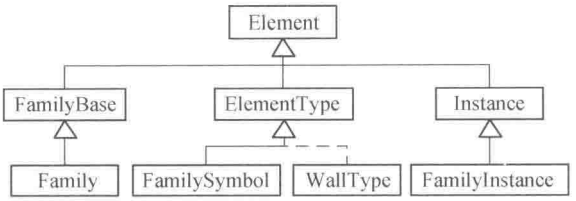


图 3-10　族，族类型，族实例类图

3.1.6　族、族类型和族实例的关系

表 3-2 列出了族、族类型、族实例三者之间相互获取关系及对应的代码实例。

表 3-2　　族、族类型、族实例三者之间的获取关系

获取关系	代码实例
从族实例获取族类型	① FamilySymbol symbol = familyInstance. Symbol; ② FamilySymbol symbol = RevitDoc. GetElement (familyInstance. GetTypeId ()) as FamilySymbol;
从族类型获取族	Family family = symbol. Family;
从族获取族类型	FamilySymbolSet setOfSymbols = family. Symbols; (返回值是一个集合，这样就获取了族下面的所有族类型)
从族类型获取族实例	FamilyInstanceFilter familyInstanceFilter = new FamilyInstanceFilter(RevitDoc, symbol. Id); FilteredElementCollector filteredElements = new FilteredElementCollector(RevitDoc); filteredElements = filteredElements. WherePasses(familyInstanceFilter); foreach (FamilyInstance element in filteredElements) { //遍历族实例 } (这里关键点是使用 FamilyInstanceFilter)

3.1.7　从族创建族实例

下面是一个族实例的创建过程，可以让用户进一步了解族、族类型和族实例之间的关系。

首先在文档中找到名字为“0762×2032 mm”的门类型，如果没有找到，则加载一个名称为“M_单-嵌板 4. rfa”的族文件，这样就得到一个 Family，然后从 Family 获取名为“0762×2032 mm”的门类型。

然后在文档中找到一个直线形的墙，计算墙的中点位置，在此位置插入类型为“0762×2032 mm”的门。

见代码片段 3-12。

代码片段 3-12：放置类型为“0762×2032 mm”的门

```
string doorTypeName = "0762 x 2032 mm";
FamilySymbol doorType = null;

//在文档中找到名字为“0762×2032 mm”的门类型
```

```
ElementFilter doorCategoryFilter = new ElementCategoryFilter(BuiltInCategory.OST_Doors);
ElementFilter familySymbolFilter = new ElementClassFilter(typeof(FamilySymbol));
LogicalAndFilter andFilter = new LogicalAndFilter ( doorCategoryFilter,
familySymbolFilter);
FilteredElementCollector doorSymbols = new FilteredElementCollector(RevitDoc);
doorSymbols = doorSymbols.WherePasses(andFilter);
bool symbolFound = false;
foreach (FamilySymbol element in doorSymbols)
{
  if (element.Name = = doorTypeName)
  {
      symbolFound = true;
      doorType = element;
      break;
  }
}

//如果没有找到,就加载一个族文件
if (! symbolFound)
{
  string file = @ "C:\ProgramData\Autodesk\RVT 2014\Libraries\Chinese_INTL\门\M_单-嵌板
  4.rfa";
  Family family;
  bool loadSuccess = RevitDoc.LoadFamily(file, out family);
  if (loadSuccess)
  {
      foreach (ElementId doorTypeId in family.GetValidTypes( ))
      {
        doorType = RevitDoc.GetElement(doorTypeId) as FamilySymbol;
        if (doorType ! = null)
        {
            if (doorType.Name = = doorTypeName)
            {
              break;
            }
        }
      }
  }
  else
  {
      Autodesk.Revit.UI.TaskDialog.Show("Load family failed", "Could not load family file '"
      + file + "'");
  }
}

//使用族类型创建门
if (doorType ! = null)
{
  //首先找到线形的墙
  ElementFilter wallFilter = new ElementClassFilter(typeof(Wall));
  FilteredElementCollector filteredElements = new FilteredElementCollector(RevitDoc);
  filteredElements = filteredElements.WherePasses(wallFilter);
  Wall wall = null;
  Line line = null;
```

```
    foreach (Wall element in filteredElements)
    {
        LocationCurve locationCurve = element.Location as LocationCurve;
        if (locationCurve != null)
        {
          line = locationCurve.Curve as Line;
          if (line != null)
          {
              wall = element;
              break;
          }
        }
    }

    //在墙的中心位置创建一个门
    if (wall != null)
    {
        XYZ midPoint = (line.get_EndPoint(0) + line.get_EndPoint(1)) / 2;
        Level wallLevel = RevitDoc.GetElement(wall.LevelId) as Level;
        //创建门:传入标高参数,作为门的默认标高
        FamilyInstance door = RevitDoc.Create.NewFamilyInstance(midPoint, doorType, wall,
        wallLevel, Autodesk.Revit.DB.Structure.StructuralType.NonStructural);
        Autodesk.Revit.UI.TaskDialog.Show("Succeed", door.Id.ToString( ));
        Trace.WriteLine("Door created: " + door.Id.ToString( ));
    }
    else
    {
        Autodesk.Revit.UI.TaskDialog.Show("元素不存在", "没有找到符合条件的墙");
    }
}
else
{
    Autodesk.Revit.UI.TaskDialog.Show("族类型不存在", "没有找到族类型'" + doorTypeName
    + "'");
}
```

注意:这里的门并不是插入到墙的中点位置,Revit® 会自动调整门的 Z 轴的值和墙平齐。另外,创建的时候,传入的标高会作为门的标高,如果不传,则门就没有标高。

结果见图 3-11。

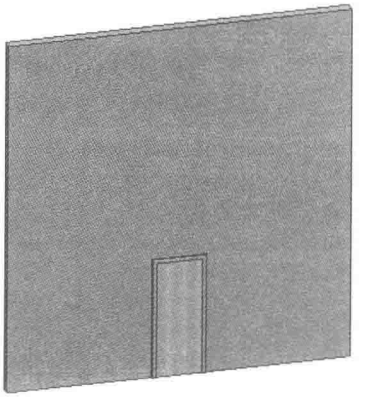


图 3-11　创建门示例

3.1.8　元素的创建

1) 创建模型元素

Revit® 创建模型元素主要有两种方式:

① 使用 Document. Create 或对应的模型元素的静态函数 Create 面,比如:

- RevitDoc. Create. NewFamilyInstance
- RevitDoc. Create. NewGrid

- RevitDoc.Create.NewDimension

② 使用 Element 子类的静态方法 Create，比如：

- Material.Create
- Wall.Create

2）创建族

代码片段 3-13 创建了一个简单的拉伸实体的族，并创建了一个族类型（详细介绍参见第 7 章）。

代码片段 3-13：创建拉伸实体族

```
//创建族文档
Document familyDoc =  RevitApp.NewFamilyDocument(@"C:\ProgramData\Autodesk\RVT 2014\
    Family Templates\Chinese\公制常规模型.rft");
using (Transaction transaction =  new Transaction(familyDoc))
{
  transaction.Start("Create family");
  CurveArray curveArray =  new CurveArray( );
  curveArray.Append(Line.CreateBound(new XYZ(0, 0, 0), new XYZ(5, 0, 0)));
  curveArray.Append(Line.CreateBound(new XYZ(5, 0, 0), new XYZ(5, 5, 0)));
  curveArray.Append(Line.CreateBound(new XYZ(5, 5, 0), new XYZ(0, 5, 0)));
  curveArray.Append(Line.CreateBound(new XYZ(0, 5, 0), new XYZ(0, 0, 0)));
  CurveArrArray curveArrArray =  new CurveArrArray( );
  curveArrArray.Append(curveArray);
  //创建一个拉伸实体
  familyDoc.FamilyCreate.NewExtrusion(true, curveArrArray, SketchPlane.Create(familyDoc,
    RevitApp.Create.NewPlane(new XYZ(0, 0, 1), XYZ.Zero)), 10);
  //创建一个族类型
  familyDoc.FamilyManager.NewType("MyNewType");
  transaction.Commit( );
  familyDoc.SaveAs("MyNewFamily.rfa");
  familyDoc.Close( );
}
```

3）创建族类型

Revit® 创建族类型主要有两种方式：

- 在创建族的时候创建类型（详细介绍参见第 7 章）。
- 使用 Symbol 的 Duplicate（复制）方法。

例如代码片段 3-14，通过复制创建一个墙类型。

代码片段 3-14：复制墙类型

```
Wall wall =  RevitDoc.GetElement(new ElementId(185521)) as Wall;
WallType wallType =  wall.WallType;
ElementType duplicatedWallType =  wallType.Duplicate(wallType.Name +  " (duplicated)");
```

3.2 元素编辑

在 Revit® 中，对元素的编辑有以下几种方法：

- Autodesk. Revit. DB. ElementTransformUtils 的移动，旋转和镜像；
- 文档类 Document 的删除；
- 创建方法 Autodesk. Revit. Creation. ItemFactoryBase 里的创建组合；
- 阵列类型的创建；
- 族编辑的一些基本类和方法。

下面将对这些方法和族编辑的基本类做详细的介绍。

3.2.1　移动

API 提供了移动元素的方法，可以把一个或者几个元素从一个地方移动到另一个地方，见表 3-3。从精确程度来说，和 UI 的移动命令是一样的。

表 3-3　　移动元素的方法

方法	描述
MoveElement(Document, ElementId, XYZ)	使用给定的平移变换移动一个元素
MoveElements(Document, ICollection<ElementId>, XYZ)	使用给定的平移变换移动元素集

MoveElement 和 MoveElements 是常用的移动方式，使用起来比较简单，但是有几个需要注意的地方：

① 移动方法不能远离标高向上或者向下移动一个基于标高的元素。也就是说，当元素是基于标高的，则不能改变 Z 轴坐标值，但可以移动元素到同一标高内的任意位置。例如，如果在坐标点(0, 0, 0)新创建了一个柱子，然后移动它到新的坐标点(10, 20, 30)，这个柱子将会移动到(10, 20, 0)，而不是(10, 20, 30)。见代码片段 3-15。

代码片段 3-15：元素移动

```
Document projectDoc = ActiveUIDocument.Document;

using(Transaction moveColumnTran = new Transaction(projectDoc, "Move a new column to the
    new place"))
{
    moveColumnTran.Start( );

    // 获取 Revit 文档的创建句柄
    Autodesk.Revit.Creation.Document creater = projectDoc.Create;
    //创建一根柱子：使用给定的位置(坐标原点)，柱子类型和标高(高度为 0)
    XYZ origin = new XYZ(0, 0, 0);
    Level level = GetALevel(projectDoc);
    FamilySymbol columnType = GetAColumnType(projectDoc);
     FamilyInstance column = creater. NewFamilyInstance (origin, columnType, level,
     Autodesk.Revit.DB.Structure.StructuralType.Column);
    // 把柱子移动到新的位置
    XYZ newPlace = new XYZ(10, 20, 30);
    ElementTransformUtils.MoveElement(projectDoc, column.Id, newPlace);

    moveColumnTran.Commit( );
}
```

② 当移动一个元素的时候，其他的元素也许会跟着移动。例如，如果一堵墙上有窗户，这堵墙移动了，窗户也会跟着移动。移动元素集的方法也会出现这种情况。例如，当移动几根柱子的时候，所有与柱子连接着的梁也会跟着被移动或者会被改变长度。

③ 如果元素被钉住，即 Pinned 属性返回值是 true，则表明这个元素不能被移动。如果仍然使用 MoveElement 方法来移动这个元素，API 就会抛出 InvalidOperationException 来提示用户不能移动被钉住的元素。

另外，还可以通过 Location 类来移动一个元素。Location 类提供了移动和旋转的方法，而且其子类提供了更多的 Location 信息和属性控制，如 LocationPoint 类和 LocationCurve 类。如果一个元素的 Location 可以转型为 LocationCurve 或者 LocationPoint，就可以直接移动这根线或者点到新的坐标点了。示例见代码片段 3-16。

代码片段 3-16：元素移动

```
Wall wall = element as Wall;
if (null != wall)
{
    LocationCurve wallLine = wall.Location as LocationCurve;
    XYZ newPlace = new XYZ(10, 20, 0);
    wallLine.Move(newPlace);
}
```

如果使用上面这种方法移动一个元素，那么请注意向量(10，20，0)并不是目标坐标值，而是一个偏移向量。图 3-12 表明了这堵墙在移动前后的位置。

另外，LocationCurve 的 Curve 属性或者 LocationPoint 的 Point 属性也可以用来移动一个元素。

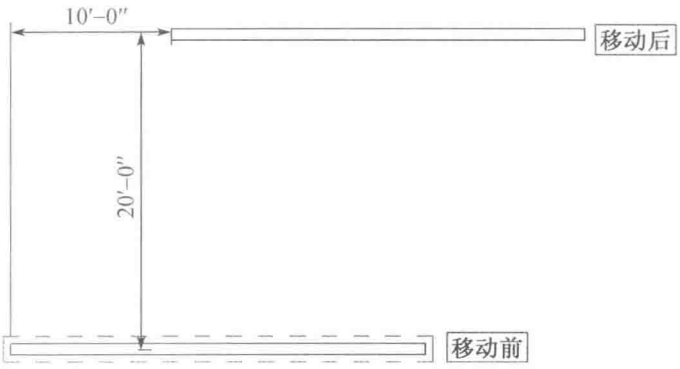


图 3-12 使用位置曲线移动一堵墙

Curve 属性可以用来移动基于曲线的元素到精确的坐标点。像这样基于曲线的元素有很多，如墙、梁、支撑等。也可以利用这个属性来改变元素的长度。示例见代码片段 3-17。

代码片段 3-17：元素移动

```
using(Transaction tran = new Transaction(projectDoc, "Change the wall's curve with a new
    location line."))
{
    tran.Start( );

    LocationCurve wallLine = wall.Location as LocationCurve;
```

```
    XYZ p1 = XYZ.Zero;
    XYZ p2 = new XYZ(10, 20, 0);
    Line newWallLine = Line.CreateBound(p1, p2);
    // 把墙的位置线换成新的线
    wallLine.Curve = newWallLine;

    tran.Commit( );
}
```

利用 LocationPoint 的 Point 属性,可以设置元素的物理位置,见代码片段 3-18。

代码片段 3-18:元素移动

```
FamilyInstance column = element as FamilyInstance;
if (null ! = column)
{
    LocationPoint columnPoint = column.Location as LocationPoint;
    XYZ newLocation = new XYZ(10, 20, 0);
    //移动柱子到新的位置(10, 20, 0)
    columnPoint.Point = newLocation;
}
```

3.2.2　旋转

API 提供了旋转方法来旋转一个或者几个元素,见表 3-4。

表 3-4　　旋转元素的方法

方法	描述
RotateElement(Document, ElementId, Line, double)	使用给定的轴线和角度对一个元素进行旋转
RotateElements(Document, ICollection<ElementId>, Line, double)	使用给定的轴线和角度对元素集进行旋转

在旋转方法中,旋转角度是用弧度计量。正值是通过轴线做逆时针的旋转,对应的负值就是做顺时针旋转,如图 3-13 和图 3-14 所示。

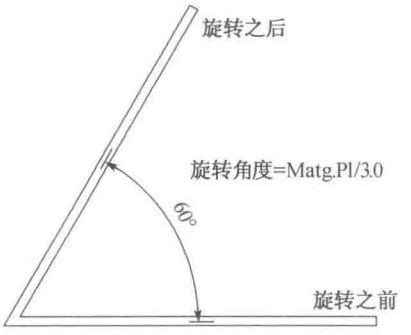


图 3-13　逆时针旋转

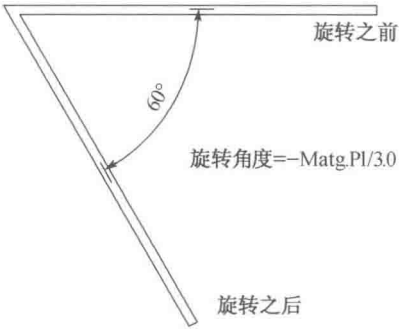


图 3-14　顺时针旋转

使用 RotateElement 和 RotateElements 方法对元素进行旋转时,需要注意下面几点:

① 当旋转一个元素时,旋转轴应该是有限线段,如果是无限线作为旋转轴线,会导致旋

转失败。

② 旋转轴一般要求与元素 LocationCurve 所在的平面垂直，否则很可能会导致旋转失败。

代码片段 3-19 获取了一堵墙，并创建了一条和墙位置线垂直的旋转轴，然后对这堵墙进行了逆时针 60°的旋转。

代码片段 3-19：元素旋转

```
using(Transaction tran = new Transaction(projectDoc, "Rotate the wall."))
{
    tran.Start( );
    LocationCurve wallLine = wall.Location as LocationCurve;
    XYZ p1 = wallLine.Curve.GetEndPoint(0);
    XYZ p2 = new XYZ(p1.X, p1.Y, 30);
    Line axis = Line.CreateBound(p1, p2);
    ElementTransformUtils.RotateElement(projectDoc, wall.Id, axis, Math.PI / 3.0);
    tran.Commit( );
    }
```

和移动元素方法一样，还有另外一种方法来旋转一个元素。如果一个元素的 Location 可以向下转型为 LocationCurve 或者 LocationPoint，那么就可以直接旋转元素对应的曲线或者点。代码片段 3-20 显示了通过墙的位置线和柱子的位置点对它们分别进行旋转。

代码片段 3-20：元素旋转

```
Document projectDoc = ActiveUIDocument.Document;

using(Transaction tran = new Transaction(projectDoc, "Rotate the wall and the column."))
{
    tran.Start( );

    Wall wall = projectDoc.GetElement(new ElementId(184163)) as Wall;

    XYZ aa = XYZ.Zero;
    XYZ cc = XYZ.Zero;
    // 通过元素的位置线来旋转元素
    LocationCurve curve = wall.Location as LocationCurve;
    if (null != curve)
    {
        Curve line = curve.Curve;
        aa = line.GetEndPoint(0);
        cc = new XYZ(aa.X, aa.Y, aa.Z + 10);
        Line axis = Line.CreateBound(aa, cc);
        curve.Rotate(axis, Math.PI / 2.0);
    }

      FamilyInstance column = projectDoc. GetElement ( new ElementId ( 184150 )) as
      FamilyInstance;

    // 通过元素的位置点来旋转元素
```

```
    LocationPoint point =  column.Location as LocationPoint;
    if (null ! =  point)
    {
        aa =  point.Point;
        cc =  new XYZ(aa.X, aa.Y, aa.Z +  10);
        Line axis =  Line.CreateBound(aa, cc);
        point.Rotate(axis, Math.PI / 3.0);
    }
    tran.Commit( );
}
```

3.2.3　镜像

Revit® API 提供了镜像方法来镜像一个或者多个元素，见表 3-5。

表 3-5　　镜像的方法

方法	描述
MirrorElement(Document, ElementId, Plane)	使用给定的平面创建一个元素的镜像拷贝
MirrorElements(Document document, ICollection＜ElementId＞ elementsToMirror, Plane plane)	使用给定的平面创建一个元素集合的镜像拷贝
CanMirrorElement(Document, ElementId)	判断元素是否可以进行镜像操作
CanMirrorElements(Document, ICollection<ElementId>)	判断元素集是否可以进行镜像操作

镜像有两个辅助方法 CanMirrorElement 和 CanMirrorElements 来判断是否可以镜像元素。代码片段 3-21 将显示如何通过一个平面来镜像一根柱子。

代码片段 3-21：元素镜像

```
using(Transaction tran =  new Transaction(projectDoc, "Mirror thecolumn."))
{
    tran.Start( );

    FamilyInstance column =  projectDoc.GetElement(new ElementId(184150)) as
    FamilyInstance;
    if (null ! =  column)
    {
      Plane plane =  new Plane(XYZ.BasisX, XYZ.Zero);
      if(ElementTransformUtils.CanMirrorElement(projectDoc, column.Id))
      {
        ElementTransformUtils.MirrorElement(projectDoc, column.Id, plane);
      }
    }
    tran.Commit( );
}
```

每个 FamilyInstance 有一个 Mirrored 属性。通过这个属性可以判断一个 FamilyInstance(如一根柱子)是否是镜像的。例如，在 3-21 示例代码中，如果这个柱子的 Mirrored 属性值是 true，则说明这是一根镜像的柱子。

注意：

① 镜像方法需要一个平面作为镜像轴，如果这个平面是 null，那么操作会失败，并且会抛 ArgumentNullException 作为提示。

② 一般应该选与 Z 平面垂直的平面作为镜像平面。

3.2.4 删除

Revit® API 文档类 Document 提供了删除一个或者多个元素的方法，见表 3-6。

表 3-6　　删除的方法

方法	描述
Delete(ElementId)	从文档中将传入 Id 的元素删除
Delete(ICollection<ElementId>)	从文档中将传入 Id 集合的元素集删除

从文档中，可以通过元素本身或者元素的 ElementId 来删除某个特定的元素。表 3-6 中的方法可以删除特定的元素或者完全依赖于这个特定元素的其他元素。被删除成功的元素会放在一个集合里返回出来，用户可以通过 ElementId 来查询这些已经删除的元素。代码片段 3-22 是对一堵墙进行删除操作。

代码片段 3-22：元素删除

```
Document projectDoc = ActiveUIDocument.Document;
    Wall wall = projectDoc.GetElement(new ElementId(184163)) as Wall;
    using(Transaction tran = new Transaction(projectDoc, "Delete the wall."))
    {
        tran.Start( );
        // 删除选择的元素：墙
        ICollection< ElementId> deletedElements = projectDoc.Delete(wall.Id);
        tran.Commit( );
    }
```

API 也提供了删除元素集合的方法来删除多个元素，请参考代码片段 3-23。

代码片段 3-23：元素删除

```
Document projectDoc = ActiveUIDocument.Document;

    List< ElementId> elementsToDelete= new List< ElementId> ( );
    using(Transaction tran = new Transaction(projectDoc, "Delete the selected elements."))
    {
        tran.Start( );
        foreach (Element elem in ActiveUIDocument.Selection.Elements)
        {
            elementsToDelete.Add(elem.Id);
        }

        ICollection< ElementId> deletedElements = projectDoc.Delete
        (elementsToDelete);
        tran.Commit( );
    }
```

注意:删除元素后,对已经删除的元素的任何引用就会变成无效的,因此如果这些引用被访问到的时候,会有异常抛出来提示用户。

3.2.5　组合

在命名空间 Autodesk. Revit. Creation 中,类 ItemFactoryBase 提供了 NewGroup(ICollection<ElementId>)方法来选择一个或者多个元素进行组合,甚至可以选择组合进行再组合,这样做可以使这些元素能在一个大的组合里面统一被修改。创建组合示例见代码片段 3-24。

代码片段 3-24:创建元素组合

```
Document projectDoc = ActiveUIDocument.Document;
List< ElementId> elementsToGroup = new List< ElementId> ( );
using(Transaction tran = new Transaction(projectDoc, "Group the selected elements."))
{
    tran.Start( );
    foreach (Element elem in ActiveUIDocument.Selection.Elements)
    {
        elementsToGroup.Add(elem.Id);
    }

    Group group = projectDoc.Create.NewGroup(elementsToGroup);
    tran.Commit( );
}
```

在新建组合的时候,通常会自动生成一个名字,比如:组 1。这个名字是可以通过修改组合类型的名字属性来改变的,请参见代码片段 3-25。

代码片段 3-25:修改元素组合类型的名字属性

```
//把默认的组合名字改成新的名字:"MyGroup"
group.GroupType.Name = "MyGroup";
```

Revit® 有两种不同的组合:模型组合和详图组合。在 API 中,两种组合都可以通过 NewGroup 方法来创建。创建的组合类型依赖于传进来的元素。如果没有任何的详图元素参与,那么模型组合就被创建。如果参与的元素包括两种类型的元素,一个模型组合和一个附着的详图组合就将被创建。

在使用组合的过程中,需要注意下面几点:

① 当元素被组合起来了,它们仍然可以被从文档中删除。在组合里面的元素被删除时,这个元素在界面上仍然是可见的,但是这个元素实际上被删除了,是不能再次被选中或者被访问的。当组合中最后一个元素被删除或者是从组合里面移除了,这个组合的实体随后也将被删除。

② 当元素在一个组合里面的时候,它们不能单独被移动或者被旋转。如果试图对这些元素进行这些操作,虽然移动或者旋转的方法返回 true,这些元素仍然不会有任何变化。

在做组合操作的时候,对于尺寸标注或者标签,如果没有对它们引用到的元素也同样做组合操作的话,是不能单独组合的。如果试图做这样的操作的话,组合操作会失败并且会抛

出如图 3-15 所示的异常。

但是，下面的操作是允许的：当组合尺寸标注和标签引用到的元素已经在一个模型组合里面的时候，可以组合它们。这样尺寸标注和标签会被加到一个附属的详图组合里。当这个附属的详图组合在其父组合没有被同样操作的话，是不能做如下所有的操作的：移动、拷贝、旋转、阵列、镜像。

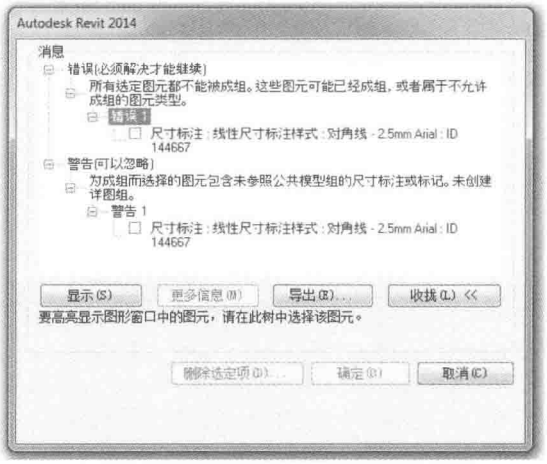


图 3-15　只组合注释而没有组合引用的异常

3.2.6　线型阵列和圆弧型阵列

Revit® API 提供的阵列方法可以对一个或者多个元素创建线型的或者是圆弧型的阵列。例如，可以选择墙上的一扇门或者一扇窗，阵列出多个门或窗的实体。表 3-7 列出了阵列方法及其描述。

表 3-7　　阵列的方法

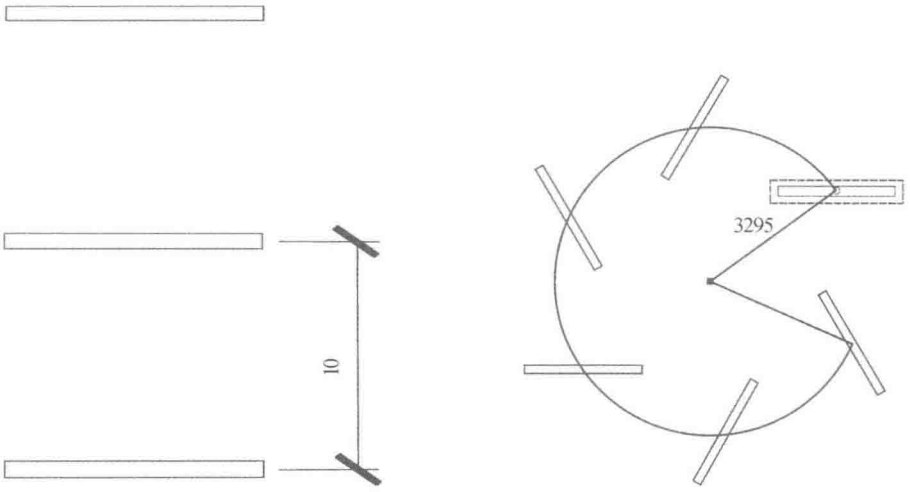
方法	描述
LinearArray. Create (Document, View, ElementId, int, XYZ, ArrayAnchorMember)	从传入的元素中创建一个线型阵列
LinearArray. Create (Document, View, ICollection＜ElementId＞, int, XYZ, ArrayAnchorMember)	从传入的元素集中创建一个线型阵列
RadialArray. Create (Document, View, ElementId, int, Line, double, ArrayAnchorMember)	从传入的元素中创建一个基于旋转轴的圆弧型阵列
RadialArray. Create (Document, View, ICollection＜ElementId＞, int, Line, double, ArrayAnchorMember)	从传入的元素集中创建一个基于旋转轴的圆弧型阵列

阵列有两种类型：线型阵列和圆弧型阵列。线型阵列是在 Revit® 中呈线型状的阵列；圆弧型阵列则呈圆弧型。代码片段 3-26 用线型阵列了一堵墙，阵列数为 3，其阵列方向为 **Y** 向量，到第二个阵列元素的平移量为 10 英尺。

代码片段 3-26：元素阵列

```
public void ArraryElements( )
    {
        Document projectDoc =  ActiveUIDocument.Document;
        Wall wall =  projectDoc.GetElement(new ElementId(2307)) as Wall;
        using(Transaction tran =  new Transaction(projectDoc, "LinearArray the wall."))
        {
            tran.Start( );
            XYZ translation =  new XYZ(0, 10, 0);
            LinearArray.Create(projectDoc, projectDoc.ActiveView, wall.Id, 3,
            translation, ArrayAnchorMember.Second);
            tran.Commit( );
        }
    }
```

上例的线型阵列如图 3-16(a)所示。如果对这堵墙创建圆弧形阵列，使用坐标 Z 轴为阵列轴线，到第二个阵列元素的旋转角度为 Math. PI/3，阵列数为 6，则效果如图 3-16(b)所示。



(a) 线型阵列　　(b) 圆弧型阵列

图 3-16　线型阵列与圆弧型阵列

创建阵列时不仅创建了模型元素，而且同时创建了对应的阵列组。阵列里面每一个实体都是阵列组合的一个成员。如图 3-16 所示的例子，阵列组的成员为原来的墙以及阵列创建的另外 2 堵墙。如果需要创建几个相同组件的实体并同时操作它们，使用阵列方法是很方便的。

使用阵列方法需要注意以下几个方面：

① 阵列数是有限定范围的，线型阵列范围是 2 到 200，圆弧型阵列是 3 到 200。

② 当做线型和圆弧型阵列操作时，其他元素依赖于这个将被阵列的元素也会跟着一起阵列。

③ 有的元素不能被阵列：如 section，因为它们不能被组合。

④ 大多数 Annotation symbols 是不支持阵列的。

LinearArray 和 RadialArray 也提供了 ArrayElementWithoutAssociate 和 ArrayElements WithoutAssociate 来阵列一个或者多个元素，这些元素是没有被组合和关联起来的，这些方法的使用类似于阵列的创建方法。

3.2.7　族编辑

Revit® 中有个很重要的概念——族(Family)。族也是一种元素，它分为系统族、内建族和构建族。系统族是 Revit® 内置的族，不可以自由编辑它的模型、类型和参数。内建族和构建族允许编辑和创建自定义的模型、类型和参数，区别是内建族只能存储在当前的项目文档里，不能被其他文档使用。本节主要介绍针对构建族进行编辑和载入的方法(详细介绍参见第 7 章)。

1) 编辑族(EditFamily)和载入族(LoadFamily)

在当前文档中，如果加载了一些构建族，并创建了这些族的族实例，那么，对某个族实例进行 EditFamily 操作，将会进入到族编辑的文档中，可以对族进行修改。这个方法是对族

在内存中新创建一个独立的拷贝文档，修改编辑好这份文档之后，如果想要之前的工程文档应用这些修改，那么需要使用带有 IFamilyLoadOptions 接口参数的 LoadFamily 重载方法重新载入族并覆盖应用到使用中的族实例中。编辑族和载入族的方法和描述请见表 3-8。

表 3-8　　编辑和载入族的方法

方法	描述
EditFamily(Family loadedFamily)	从传入的构建族中获得要修改的族文档
LoadFamily (Document targetDocument, IFamilyLoadOptions familyLoadOptions)	将族文档的内容载入到目标文档中

Autodesk. Revit. DB. IFamilyLoadOptions 是提供族载入选项的接口类，它有两个接口函数，可以根据需要实现这两个接口函数，在 LoadFamily 中就可以实现想要的功能，见表 3-9。

表 3-9　　IFamilyLoadOptions 的接口类方法

方法	描述
OnFamilyFound (bool familyInUse, out bool overwriteParameterValues)	当族实例在目标文档中被找到时，将被触发执行的回调函数
OnSharedFamilyFound(Family sharedFamily, bool familyInUse, out FamilySource source, out bool overwriteParameterValues)	当共享族实例在目标文档中被找到时，将被触发执行的回调函数

代码片段 3-27 简单介绍了如何使用 EditFamily，LoadFamily 方法以及接口。

首先实现 IFamilyLoadOptions 接口的两个方法，将覆盖参数值选项 overwriteParameterValues 设为 true，作用是在重新载入族的时候会覆盖旧的参数和参数值。这个类的实例将会在后面的代码 LoadFamily 方法中使用到。

代码片段 3-27：元素编辑

```
class projectFamLoadOption: IFamilyLoadOptions
{
    bool IFamilyLoadOptions.OnFamilyFound(bool familyInUse, out bool
    overwriteParameterValues)
    {
        overwriteParameterValues = true;
        return true;
    }
    bool IFamilyLoadOptions.OnSharedFamilyFound(Family sharedFamily, bool familyInUse,
    out FamilySource source, out bool overwriteParameterValues)
    {
        source = FamilySource.Project;
        overwriteParameterValues = true;
        return true;
    }
}
```

然后在当前工程文档中编辑一个构建族，如门、窗、桌子等。在族文档中加入一个新的参数——文字类型，名字为“MyParam”。修改后将编辑的族重新载入到工程文档，这样该

族所有的族实例都会有一个名字为“MyParam”的新参数了。请参考代码片段 3-28。

代码片段 3-28:元素编辑

```
Document projectDoc =  ActiveUIDocument.Document;
//这里是自定义族实例,比如门,窗,桌子…
        FamilyInstance famInst =  elem as FamilyInstance;
//编辑族,拿到族文档
Document familyDoc =  projectDoc.EditFamily(famInst.Symbol.Family);
    // 在族文档中添加一个新的参数
    using(Transaction tran =  new Transaction(projectDoc, "Edit family Document."))
    {
        tran.Start( );
        string paramName =  "MyParam ";
       familyDoc. FamilyManager. AddParameter (paramName, BuiltInParameterGroup. PG_TEXT,
       ParameterType.Text, false);
        tran.Commit( );
    }
//将这些修改重新载入到工程文档中
Family loadedFamily =  familyDoc.LoadFamily(RevitDoc, new projectFamLoadOption( ));
```

在编辑族和载入族的时候,需要注意下面几点:

① 如果当前文档正在被修改中(有打开没有被关掉的事务)或者处在只读的状态,EditFamily 方法不能被调用。可以使用 IsModifiable 和 IsReadOnly 属性来判断和检查文档的状态。

② EditFamily 方法也不能在动态更新机制(Dynamic Updates)中使用。

③ 在 LoadFamily 方法,可以用 RevitUIFamilyLoadOptions 弹出一个提示框来与用户进行交互。

2) 参照平面(Referenceplane)和参照线(Reference Line)

Referenceplane 是参照平面,Reference Line 是参照线。它们是辅助绘图的重要工具,在族的创建过程中经常使用。

参照平面可以定位和驱动族模型。在大多数的族模板中,已经默认有三个参照平面,分别为 X,Y 和 Z 平面方向,其交点是坐标原点(0, 0, 0)。这三个参照平面默认被固定锁住,并且不能被删除。一般情况下不要去解锁和移动这三个参照平面,否则可能导致所创建族原点不在(0, 0, 0)点,无法在项目文档中正确使用。参照平面也通常与参数标注结合使用,将实体的一面与参照平面对齐并锁住,它就能驱动实体模型了。

创建参照平面有两个重载方法,在 Autodesk. Revit. Creation. ItemFactoryBase 实用类里面,方法描述如表 3-10 所示。

表 3-10　　创建参照平面方法

方法	描述
NewReferencePlane (XYZ bubbleEnd, XYZ freeEnd, XYZ cutVec, View pView)	使用两点、切向量和视图创建一个新的参考平面的实例。切向量应该垂直于两点向量
NewReferencePlane2 (XYZ bubbleEnd, XYZ freeEnd, XYZ thirdPnt, View pView)	使用三点和视图创建一个新的参考平面的实例

代码片段 3-29 演示了在族文档中创建了一个名字为“MyReferencePlane”的参照平面。

代码片段 3-29:元素编辑

```
public void CreatReferencePlane( )
{
    Document doc =  this.ActiveUIDocument.Document;
    if(! doc.IsFamilyDocument)
        return;

    using(Transaction transaction =  new Transaction(doc, "Editing Family"))
    {
        transaction.Start( );
        XYZ bubbleEnd =  new XYZ(0, 5, 5);
        XYZ freeEnd =  new XYZ(5, 5, 5);
        XYZ cutVector =  XYZ.BasisY;
        View view =  doc.ActiveView;
        ReferencePlane referencePlane =  doc.FamilyCreate.NewReferencePlane
        (bubbleEnd, freeEnd, cutVector, view);
        referencePlane.Name =  "MyReferencePlane";
        transaction.Commit( );
    }
}
```

注意:参照平面的名字不能重复,但是可以重命名。

图 3-17 是示例代码创建出来的参照平面。创建好之后,可以得到一些在图形建模中比较常用的数据,如方向(Direction)、法向量(Normal)、平面(Plane),以及其几何引用(Reference)等。

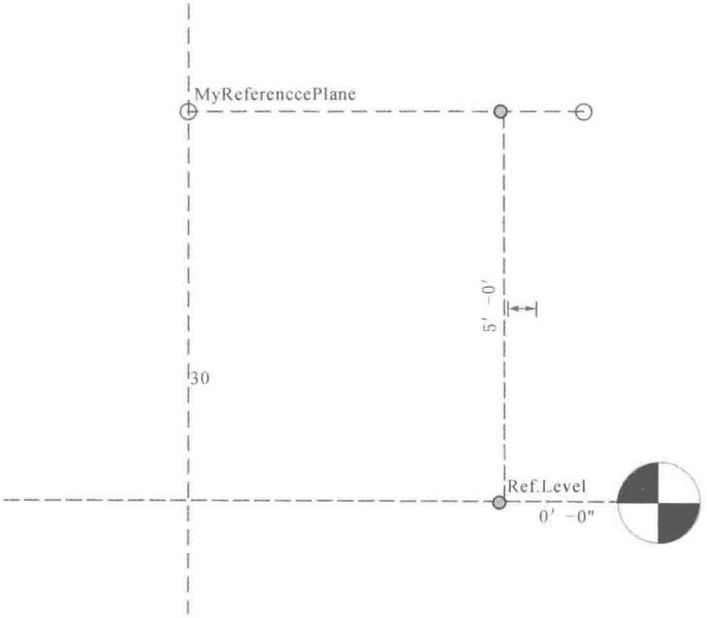


图 3-17　参照平面

参照线和参考平面的功能基本相同,它主要实现角度参变。参照线的形状既可以是直线,也可以是其他曲线,在 API 中,可以使用 ChangeToReferenceLine 方法从模型曲线转变成参照线,见代码片段 3-30。

代码片段 3-30:元素编辑

```
public void ChangeModelCurveToReferenceLine( )
    {
        Document doc =  this.ActiveUIDocument.Document;
        ModelCurve modelCurve =  doc.GetElement(new ElementId(2910)) as ModelCurve;
        using(Transaction transaction =  new Transaction(doc, "Change model curve to
        reference line."))
        {
            transaction.Start( );
            modelCurve.ChangeToReferenceLine( );
            transaction.Commit( );
        }
    }
```

上面的示例中,ID 为 2910 的元素是当前族文档中的一根模型弧线,把它变成参照线后如图 3-18 所示,图上半部分的线为原来的模型曲线,下半部分的线为转变后的参照线。

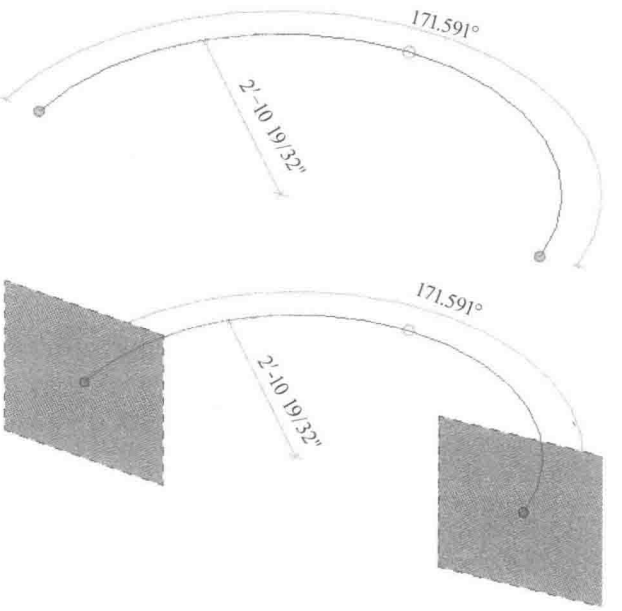


图 3-18　模型曲线与参照线

虽然参照线是由模型曲线转变来的,但是参照线仍然是模型曲线,它的元素 ID 也没有改变。可以从文档中把模型曲线过滤出来,通过查看它的 IsReferenceLine 属性,可以发现参照线的这个属性返回的是 True,即为参照线。

3) 草图平面(Sketchplane)和草图(Sketch)

草图编辑在 Revit® 中被广泛用于创建各种元素,包括绘制和修改模型的形状。没有用于创建元素的绘制形状的绘画也属于草图编辑,例如,用鼠标移动的操作画一根曲线,在已经创建好的楼板上绘制褶皱来修改楼板的几何形状等。

草图编辑在用户界面上有专门的窗口区域,不同的元素提供不同的绘制草图的工具集,标准的工具集一般有两种:

- 绘制形状的工具——直线,曲线,圆,多边形,椭圆,样条曲线。
- 拾取已有几何图形来创建形状的工具——线,面拾取等。

图 3-19 为模型线的绘制草图工具集。

草图编辑需要在一个表面上绘制形状，当这个表面为平面时称之为草图平面(SketchPlane)。草图平面也可以理解成以下几种情形：

- 用户界面的工作平面或者详图元素的视图平面。
- 标高或者其他的由草图编辑工具所决定的平面。
- 元素约束条件所决定的表面。
- 鼠标光标所在区域的一个几何面。

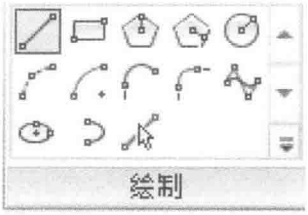


图 3-19 绘制工具集

草图平面在 API 中被用于创建草图相关的元素，如模型线；或者用于创建拥有草图属性的元素，如构建的实体。草图平面可以从已存在的元素或者从几何面、平面创建。

注意：

① 创建元素时传入的草图平面未必就是这个元素的实际草图平面，如果传入的草图平面已经被其他用途使用了，Revit®就会寻找或者创建一个新的在几何上对等的草图平面。

② 一些草图平面只适合于用在创建详图元素上，比如那些从详图曲线获取到的草图平面，当被用在其他类型的元素上时，将会导致创建失败。

创建草图平面的方法在 Autodesk. Revit. DB. SketchPlane 类里，三个重载方法如表 3-11所示。

表 3-11　　创建草图平面的方法

方法	描述
Create(Document document, ElementId datumId)	从网格、引用平面或者标高等元素创建一个草图平面
Create(Document document, Plane plane)	从一个几何平面创建一个草图平面
Create(Document document, Reference planarFaceReference)	从一个几何平面的引用创建一个草图平面

代码片段 3-31 从文档中找到名字为“Ref. Level”的标高，以此创建了一个新的草图平面，然后用几何点为(0, 0, 0)到点(10, 10, 0)的直线创建了一条模型曲线，见图 3-20。

代码片段 3-31：元素编辑

```
public void CreateModelCurve( )
    {
        Document doc = this.ActiveUIDocument.Document;
        // 在族文档中找到名字为"Ref. Level"的标高
        FilteredElementCollector collector = new FilteredElementCollector(doc);
        collector = collector.OfCategory(BuiltInCategory.OST_Levels);
        var levelElements = from element in collector where element.Name = = "Ref. Level"
        select element;
        List< Autodesk.Revit.DB.Element> levels = levelElements.ToList< Autodesk.Revit.
        DB.Element> ( );
        if(levels.Count < = 0)
            return;
        Level refLevel = levels[0] as Level;

        //创建一条几何直线，一个基于标高的草图平面，然后在这个草图平面上创建一条模型线.
        using(Transaction trans = new Transaction(doc, "Create model line."))
```

```
        {
            trans.Start( );
            Line line =  Line.CreateBound(XYZ.Zero, new XYZ(10, 10, 0));
            SketchPlane sketchPlane =  SketchPlane.Create(doc, refLevel.Id);
            ModelCurve modelLine =  doc.FamilyCreate.NewModelCurve(line, sketchPlane);
            trans.Commit( );
    }
}
```

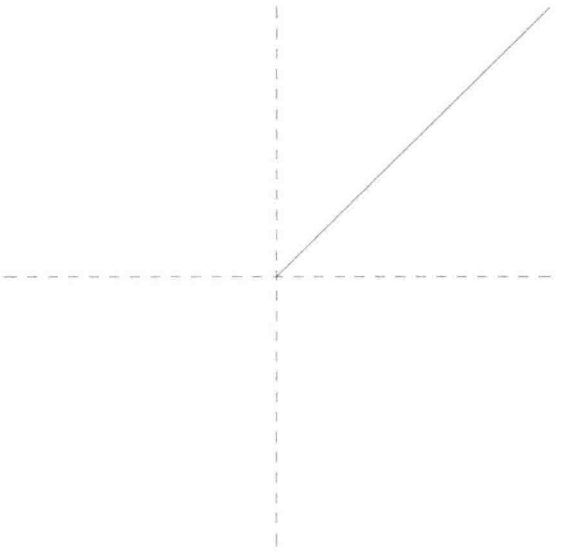


图 3-20　在草图平面上创建模型曲线(直线)

代码片段 3-32 创建一个原点为坐标原点(0, 0, 0),法向量为(0, 0, 1)的几何平面,然后用这个几何平面创建了对应的草图平面,再在这个草图平面上创建了一个圆心为(0, 0, 0),半径为 5 的圆形的模型线。

代码片段 3-32:元素编辑

```
public void CreateSketchPlaneByPlane( )
    {
        Document doc =  this.ActiveUIDocument.Document;
        using(Transaction trans =  new Transaction(doc, "Create model arc."))
        {
            trans.Start( );
            Plane plane =  this.Application.Create.NewPlane(XYZ.BasisZ, XYZ.Zero);
            SketchPlane sketchPlane =  SketchPlane.Create(doc, plane);

            Arc arc =  Arc.Create(plane, 5, 0, Math.PI *  2);
            ModelCurve modelCircle =  doc.FamilyCreate.NewModelCurve(arc, sketchPlane);
            trans.Commit( );
    }
}
```

代码的效果如图 3-21 所示。

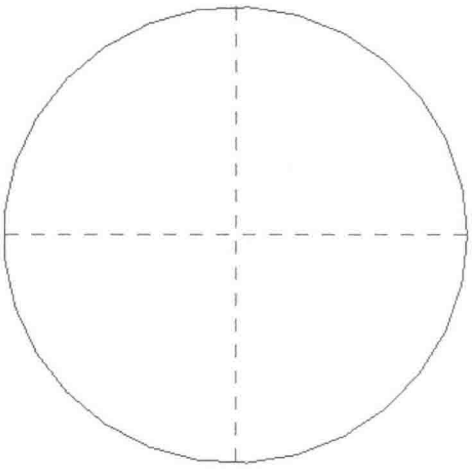


图 3-21　在草图平面上创建模型曲线(圆)

从一个几何平面的引用来创建草图平面的方法和上面两个示例类似，需要在文档中找到一个几何平面，例如，一个几何实体的平面(PlanarFace. Reference)，然后将该平面的引用传入即可。

草图(Sketch)是个抽象的概念，由两个最基本的属性构成：草图平面(SketchPlane)和轮廓(Profile)。在 UI 用户界面上，进入草图编辑的绘图区域后，草图平面就默认设置好了，用户只需要使用草图编辑工具绘制轮廓就可以；API 中则需要用户创建或者确定草图平面，再提供相应的轮廓。上面的两个示例中的端点为(0，0，0)和(10，10，0)的直线(line)；圆心为(0，0，0)和半径为 5 的圆形(arc)就可以理解为对应的轮廓。

图 3-22 左边部分是族文档中的一个拉伸体(Extrusion)，右边部分是这个拉伸体的轮廓，为四根首尾相连的直线构成的线组。

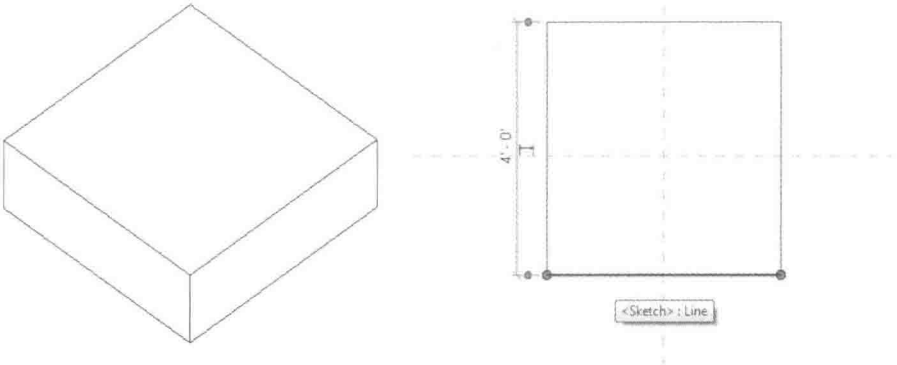


图 3-22　拉伸体及其轮廓

API 从文档中取到 ID 为 3388 的拉伸体及其草图的示例参见代码片段 3-33。

代码片段 3-33：元素编辑

```
public void GetSketchFromExtrusion( )
    {
        Document doc =  this.ActiveUIDocument.Document;
        Extrusion extrusion =  doc.GetElement(new ElementId(3388)) as Extrusion;
        SketchPlane sketchPlane =  extrusion.Sketch.SketchPlane;
        CurveArrArray sketchProfile =  extrusion.Sketch.Profile;
    }
```

Revit® 中的三维模型，如拉伸(Extrusion)、融合(Blend)、旋转(Revolution)、放样(Sweep)、放样融合(SweptBlend)，都使用到了草图。不但在创建好的模型中可以获取到它的草图(Sketch 属性)，而且在创建过程中(比如 NewExtursion 等方法)也需要使用到草图平面和轮廓。关于三维模型的创建和使用方法(详细介绍参见第 7 章)。

3.3　元素过滤器(ElementFilters)

Revit® API 提供了元素过滤器，用来遍历元素以及元素的 ID。这些过滤器为各种不同的应用程序获取元素提供了更灵活更实用的接口。

3.3.1　收集器(FilteredElementCollector)

用来迭代以及过滤元素的主要类是收集器，它有三种构造方式，见表 3-12，可以根据具体情况使用任意一种。

表 3-12　　收集器的三种构造函数

构造函数	描述
FilteredElementCollector(Document document)	从一个文档构造，迭代会从文档所有的元素进行
FilteredElementCollector(Document document, ICollection<ElementId> elementIds)	从一个文档和 ElementID 集合构造，迭代会从这传进来的集合中进行
FilteredElementCollector(Document document, ElementId viewId)	从一个文档和一个视图构造，迭代会从所有在传进来的视图中所见的元素中进行

注意：*初次创建该对象的时候，是没有应用任何的过滤器的，如果这时试图从该对象中迭代或者获取元素列表，会有异常抛出。*

FilteredElementCollector 提供了一系列的方法，允许用户设置查询和过滤感兴趣的元素集：

- 通用方法 WherePasses()，在收集器中应用单一的 ElementFilter。该方法可以在结果中重复使用来增加不同的过滤器(filter)。
- 快捷方法，不需要传初始化过滤器对象的易于使用的方法，相当于常用过滤器 filter 的快捷键。比如 OfClass()，OfCategoryId()，OwnedByView()。
- 并集、交集等运算方法，比如 UnionWith()和 IntersectWith()。

这些方法返回收集器本身，并允许不同的过滤器链式调用。示例参见代码片段 3-34。

代码片段 3-34：元素过滤器

```
FilteredElementCollector collection =  new FilteredElementCollector(RevitDoc);
ElementFilter filter =  new ElementCategoryFilter(BuiltInCategory.OST_StackedWalls);
collection.OfClass(typeof(Wall)).WherePasses(filter);
ICollection< ElementId>  foundIds =  collection.ToElementIds( );
```

向收集器设置完必须的的过滤器之后，就可以获取符合条件的元素了，有以下几种获取方式：

- 获取元素 ID 或者元素本身的迭代器。
- 获取符合过滤器条件的元素 ID 或者元素本身的集合。

• 获取符合过滤器条件的第一个元素 ID 或者元素本身。当只需要唯一一个符合条件的元素时，这种方式很有用，因为不用再去遍历所有符合条件的元素了。

1）foreach 语法支持

FilteredElementCollector 支持 foreach 语法，代码片段 3-35 查询并遍历文档中所有的标高。

代码片段 3-35：元素过滤器

```
FilteredElementCollector collector =  new FilteredElementCollector(m_doc);
  //查询并遍历文档中所有的 Level
     collector. WherePasses (new  ElementCategoryFilter (BuiltInCategory. OST _ Levels )).
    WhereElementIsNotElementType( );
    foreach(Level level in collector)
{
    TaskDialog.Show("Level Name", level.Name);
}
```

2）LINQ 支持

FilteredElementCollector 类对元素实现了 IEnumerable 接口，因此，可以对这个类使用 LINQ 查询语句和操作。参见代码片段 3-36。

代码片段 3-36：元素过滤器

```
FilteredElementCollector collector =  new FilteredElementCollector(m_doc);

//首先使用一个内建的过滤器来减少后面使用 LINQ 查询的元素数量
collector.WherePasses(new ElementCategoryFilter(BuiltInCategory.OST_Levels));

// LINQ 查询：找到名字为“Level 1”的标高
var levelElements =  from element in collector
                     where element.Name = =  "Level 1"
                     select element;
List< Autodesk. Revit. DB. Element >  levels =  levelElements. ToList < Autodesk. Revit. DB.
Element> ( );

ElementId level1Id =  levels[0].Id;
```

3.3.2 元素过滤器的层次

元素过滤器是一个检验一个元素是否符合某种标准的类。它被分为三个层次：

• 快速过滤器（Quickfilters）：只检查记录的元素并防止元素在内存中展开。这种过滤器可以提高迭代的效率并且减少内存的消耗。

• 慢速过滤器（Slowfilters）：先在内存中获取和展开元素，再进行查询，因此在效率上会比较低。推荐的方法是慢速过滤器和快速过滤器结合起来用，这样会减少展开元素的数量。

• 逻辑过滤器（Logicalfilters）：由两个及以上的过滤器逻辑组成的过滤器。

注意：无论应用的条件和过滤器的顺序是什么，Revit® 内部会对过滤器重新排序来减少

元素的展开。

3.3.3　内建过滤器(Built-In filters)

为了方便使用，Revit® 提供了一些预制的过滤器，有些同时提供了FilteredElementCollector 的快捷方法，如表 3-13 所示。

表 3-13　　内建过滤器

内建过滤器	类型	符合条件值	快捷方法
LogicalAndFilter	逻辑	必须符合两个或两个以上的过滤器的元素	WherePasses()：添加另外一个过滤器； IntersectWith()：合并两个独立的过滤器的集合
LogicalOrFilter	逻辑	必须符合两个或两个以上的过滤器中的其中一个过滤器的元素	UnionWith()：合并两个独立的过滤器的集合
ElementCategoryFilter	快	符合传入类别 ID 的元素	OfCategoryId()
ElementMulticategoryFilter	快	符合任意一个传入类别 ID 的元素	无
ElementClassFilter	快	符合传入类的元素	OfClass()
ElementMulticlassFilter	快	符合任意一个传入类的元素	无
ElementIsElementTypeFilter	快	是元素类型的元素	WhereElementIsElementType(), WhereElementIsNotElementType()
ElementOwnerViewFilter	快	是视图相关的元素	OwnedByView(), WhereElementIs ViewIndependent()
ElementDesignOptionFilter	快	符合在特定设定选项的元素	ContainedInDesignOption()
ElementIsCurveDrivenFilter	快	是由线驱动的元素	WhereElementIsCurveDriven()
ElementStructuralTypeFilter	快	符合给定结构类型的元素	无
ElementWorksetFilter	快	符合在给定工作集里面的元素	无
FamilySymbolFilter	快	特定的族类型	无
ExclusionFilter	快	除传入过滤器的元素外的所有元素	Excluding()
BoundingBoxIntersectsFilter	快	其边界盒与给定的轮廓(outline)相交的元素	无
BoundingBoxIsInsideFilter	快	其边界盒在给定轮廓(outline)内的元素	无
BoundingBoxContainsPointFilter	快	其边界盒包含了给定点的元素	无
FamilyInstanceFilter	慢	特定族类型的实例	无
ElementLevelFilter	慢	和指定标高(level)联系的元素	无
ElementPhaseStatusFilter	慢	和指定阶段状态联系的元素	无
ElementParameterFilter	慢	符合一个或多个参数值的元素	无
ElementIntersectsElementFilter	慢	使用冲突检测来查找符合与传入的元素在几何上相交的元素	无
ElementIntersectsSolidFilter	慢	使用接口检测来查找符合与传入的实体在几何(solid)上相交的元素	无
PrimaryDesignOptionMemberFilter	慢	主选项所拥有的元素	无

续表

内建过滤器	类型	符合条件值	快捷方法
StructuralInstanceUsageFilter	慢	属于传入的结构实例用法(StructuralInstanceUsage)的族实例	无
StructuralWallUsageFilter	慢	属于传入结构墙用法(StructuralWallUsage)的墙	无
StructuralMaterialTypeFilter	慢	属于传入结构材料类型(StructuralMaterialType)的族实例	无
RoomFilter	慢	房间元素	无
SpaceFilter	慢	空间元素	无
AreaFilter	慢	面积元素	无
RoomTagFilter	慢	房间标签元素	无
SpaceTagFilter	慢	空间标签元素	无
AreaTagFilter	慢	面积标签元素	无
CurveElementFilter	慢	属于或者不属于(inverted)传入线类型的线型元素	无

3.3.4 元素快速过滤器(ElementQuickFilter)

下面介绍一些常用的快速过滤器的用法。

1) ElementCategoryFilter

ElementCategoryFilter 是使用类别(Category)来查询元素的过滤器。代码片段 3-37 使用墙的类别(OST_Walls)来查询当前文档的元素,文档中所有的墙实例以及墙类型都将会被过滤出来。

代码片段 3-37:元素过滤器

```
/// < summary>
///使用 ElementCategoryFilter 过滤元素
/// < /summary>
void TestElementCategoryFilter(Document doc)
{
    //找到所有属于墙类别的元素:墙实例和墙类型都将会被过滤出来
    FilteredElementCollector collector =  new FilteredElementCollector(doc);
    ElementCategoryFilter filter =  new ElementCategoryFilter(BuiltInCategory.OST_
    Walls);
    ICollection< Element>  founds =  collector.WherePasses(filter).ToElements( );
    foreach (Element elem in founds)
    {
        Trace.WriteLine(String.Format("Element id: {0}, type: {1}", elem.Id.IntegerValue,
        elem.GetType( ).Name));
    }
}
```

2) ElementClassFilter

ElementClassFilter 是使用类来查询元素的过滤器,它将会精确匹配传入的类或者该类

的子类。代码片段 3-38 使用族类型(FamilySymbol)来查询当前文档的元素,所有属于族类型的元素都将会被过滤出来。

代码片段 3-38:元素过滤器

```
/// < summary>
///使用 ElementClassFilter 过滤元素
/// < /summary>
void TestElementClassFilter(Document doc)
{
    //找到所有属于 FamilySymbol 的元素:元素的子类也将被过滤出来
    FilteredElementCollector collector =  new FilteredElementCollector(doc);
    ElementClassFilter filter =  new ElementClassFilter(typeof(FamilySymbol));
    ICollection< ElementId>  founds =  collector.WherePasses(filter).ToElementIds( );
    Trace.WriteLine(String.Format("Found {0} FamilySymbols.", founds.Count));
}
```

注意:ElementClassFilter 不支持 API 中一小部分的元素子类。这是由于这些类只存在于 API 中,Revit® 并没有对应的内部模型对象。因此,如果想用类过滤器来查找这些不支持的类,需要使用更高一级的父类来过滤,并且对过滤出来的元素进行更进一步的匹配来查找这些子类。下面列出的是 API 受到这个限制影响且并不支持的类型。

- Autodesk. Revit. DB. Material 子类
- Autodesk. Revit. DB. CurveElement 子类
- Autodesk. Revit. DB. ConnectorElement 子类
- Autodesk. Revit. DB. HostedSweep 子类
- Autodesk. Revit. DB. Architecture. Room 子类
- Autodesk. Revit. DB. Mechanical. Space 子类
- Autodesk. Revit. DB. Area 子类
- Autodesk. Revit. DB. Architecture. RoomTag 子类
- Autodesk. Revit. DB. Mechanical. SpaceTag 子类
- Autodesk. Revit. DB. AreaTag 子类
- Autodesk. Revit. DB. CombinableElement 子类
- Autodesk. Revit. DB. Mullion 子类
- Autodesk. Revit. DB. Panel 子类
- Autodesk. Revit. DB. AnnotationSymbol 子类
- Autodesk. Revit. DB. Structure. AreaReinforcementType 子类
- Autodesk. Revit. DB. Structure. PathReinforcementType 子类
- Autodesk. Revit. DB. AnnotationSymbolType 子类
- Autodesk. Revit. DB. Architecture. RoomTagType 子类
- Autodesk. Revit. DB. Mechanical. SpaceTagType 子类
- Autodesk. Revit. DB. AreaTagType 子类
- Autodesk. Revit. DB. Structure. TrussType 子类

3) ElementIsElementTypeFilter

ElementIsElementTypeFilter 是用来匹配元素类型的过滤器，可以过滤出文档中所有的元素类型的元素或者过滤出所有非元素类型的元素。代码片段 3-39 使用默认的构造函数来过滤出文档的所有元素类型。

代码片段 3-39：元素过滤器

```
/// < summary>
///使用 ElementIsElementTypeFilter 过滤元素
/// < /summary>
void TestElementIsElementTypeFilter(Document doc)
{
    //找到所有属于 ElementType 的元素
    FilteredElementCollector collector =  new FilteredElementCollector(doc);
    ElementIsElementTypeFilter filter =  new ElementIsElementTypeFilter( );
    ICollection< ElementId>  founds =  collector.WherePasses(filter).ToElementIds( );
    Trace.WriteLine(String.Format("Found {0} ElementTypes.", founds.Count));
}
```

注意：也可以使用传参的构造函数来得到所有不是元素类型的元素。

4) FamilySymbolFilter

FamilySymbolFilter 是使用传入的族来得到其所有族类型的过滤器。代码片段 3-40 取得了文档的所有的族，并且使用 FamilySymbolFilter 得到了所有的族类型。

代码片段 3-40：元素过滤器

```
/// < summary>
///使用 FamilySymbolFilter 过滤元素
/// < /summary>
void TestFamilySymbolFilter(Document doc)
{
    //找到当前文档中族实例所对应的族类型
    FilteredElementCollector collector =  new FilteredElementCollector(doc);
    ICollection< ElementId>  famIds =  collector.OfClass(typeof(Family)).ToElementIds( );
    foreach (ElementId famId in famIds)
    {
        collector =  new FilteredElementCollector(doc);
        FamilySymbolFilter filter =  new FamilySymbolFilter(famId);
        int count =  collector.WherePasses(filter).ToElementIds( ).Count;
        Trace.WriteLine(String.Format("{0} FamilySybmols belong to Family {1}.", count,
        famId.IntegerValue));
    }
}
```

5) ExclusionFilter

ExclusionFilter 是自动排除元素集的过滤器。代码片段 3-41 使用所有族类型作为排除的集合，对文档中的元素类型进行排除，得到一个没有族类型的元素类型集合。

代码片段 3-41:元素过滤器

```
/// < summary>
///使用 ExclusionFilter 过滤元素
/// < /summary>
void TestExclusionFilter(Document doc)
{
    //找到所有除族类型 FamilySymbol 外的元素类型 ElementType
    FilteredElementCollector collector = new FilteredElementCollector(doc);
    ICollection< ElementId> excludes = collector.OfClass(typeof(FamilySymbol)).
    ToElementIds( );

    //创建一个排除族类型 FamilySymbol 的过滤器
    ExclusionFilter filter = new ExclusionFilter(excludes);
    ICollection< ElementId> founds = collector.WhereElementIsElementType( ).
    WherePasses(filter).ToElementIds( );
    Trace.WriteLine(String.Format("Found {0} ElementTypes which are not FamilySybmols",
    founds.Count));
}
```

3.3.5　元素慢速过滤器(ElementSlowFilter)

下面介绍一些常用的慢速过滤器的用法。

1) ElementLevelFilter

ElementLevelFilter 是使用关联的标高(Level)来匹配其关联或不关联的元素。代码片段 3-42 取得了文档中每一个标高所关联的元素。

代码片段 3-42:元素过滤器

```
/// < summary>
///使用 ElementLevelFilter 过滤元素
/// < /summary>
void TestElementLevelFilter(Document doc)
{
    //找到当前所有标高对应的所有元素
    FilteredElementCollector collector = new FilteredElementCollector(doc);
    ICollection< ElementId> levelIds = collector.OfClass(typeof(Level)).ToElementIds
( );
    foreach (ElementId levelId in levelIds)
    {
        collector = new FilteredElementCollector(doc);
        ElementLevelFilter filter = new ElementLevelFilter(levelId);
        ICollection< ElementId> founds = collector.WherePasses(filter).ToElementIds( );
        Trace.WriteLine(String.Format("{0} Elements are associated to Level {1}.",
        founds.Count, levelId.IntegerValue));
    }
}
```

注意:可以使用 new ElementLevelFilter(levelId,true)来取得与传入标高不关联的所有元素。

2）ElementParameterFilter

ElementParameterFilter 是使用一个或者多个参数过滤规则来匹配元素的过滤器。代码片段 3-43 构造了一个使用元素 ID 值来匹配的规则，查找文档中所有 ID 值大于 99 的元素集。

代码片段 3-43：元素过滤器

```
/// < summary>
///使用 ElementParameterFilter 过滤元素
/// < /summary>
void TestElementParameterFilter(Document doc)
{
    //找到所有 id 大于 99 的元素
    BuiltInParameter testParam = BuiltInParameter.ID_PARAM;
    //提供者
     ParameterValueProvider pvp = new ParameterValueProvider (new ElementId ((int)
     testParam));
    //评估者
    FilterNumericRuleEvaluator fnrv = new FilterNumericGreater( );
    //规则值
    ElementId ruleValId = new ElementId(99); // Id 大于 99
    //创建规则过滤器及对应的元素参数过滤器
    FilterRule fRule = new FilterElementIdRule(pvp, fnrv, ruleValId);
    ElementParameterFilter filter = new ElementParameterFilter(fRule);
    FilteredElementCollector collector = new FilteredElementCollector(doc);
    ICollection< Element> founds = collector.WherePasses(filter).ToElements( );
    foreach (Element elem in founds)
    {
        Trace.WriteLine(String.Format("Element id: {0}", elem.Id.IntegerValue));
    }
}
```

注意：

① ElementParameterFilter 也提供了 inverted 参数，用来匹配不符合传入规则的元素集。

② 如果参数含有单位（如长度、面积、体积等），该单位应当使用内部单位。比如说面积的内部单位是平方英尺（SF）。

3）FamilyInstanceFilter

FamilyInstanceFilter 是使用族类型来匹配对应族实例元素的过滤器。代码片段 3-44 查找文档中族类型名字为"W10X49"对应的所有族实例。

代码片段 3-44：元素过滤器

```
/// < summary>
///使用 FamilyInstanceFilter 过滤元素
/// < /summary>
void TestFamilyInstanceFilter(Document doc)
```

```
{
    //找到名字"W10X49"的族类型
    FilteredElementCollector collector = new FilteredElementCollector(Document);
    collector = collector.OfClass(typeof(FamilySymbol));
    var query = from element in collector
        where element.Name = = "W10X49"
        select element; // Linq 查询
    List< Autodesk.Revit.DB.Element> famSyms = query.ToList< Autodesk.Revit.DB.
    Element> ( );
    ElementId symbolId = famSyms[0].Id;

    //创建过滤器并找到该族类型对应的所有族实例
    collector = new FilteredElementCollector(doc);
    FamilyInstanceFilter filter = new FamilyInstanceFilter(doc, symbolId);
    ICollection< Element> founds = collector.WherePasses(filter).ToElements( );
    foreach (FamilyInstance inst in founds)
    {
        Trace.WriteLine(String.Format("FamilyInstance {0}, FamilySybmol Id {1}, Name: {2}",
        inst.Id.IntegerValue, inst.Symbol.Id.IntegerValue, inst.Symbol.Name));
    }
}
```

4) CurveElementFilter

CurveElementFilter 是匹配线型元素的过滤器。代码片段 3-45 用线元素类型(CurveElementType)来匹配对应的线型元素,如果想找模型线、参考线等线型元素,使用这个过滤器会比较方便。

代码片段 3-45:元素过滤器

```
/// /// < summary>
///使用 CurveElementFilter 过滤元素
/// < /summary>
void TestCurveElementFilter(Document doc)
{
    //找到所有线元素类型对应的线型元素
    Array stTypes = Enum.GetValues(typeof(CurveElementType));
    foreach (CurveElementType tstType in stTypes)
    {
        if (tstType = = CurveElementType.Invalid) continue;
        FilteredElementCollector collector = new FilteredElementCollector(Document);
        CurveElementFilter filter = new CurveElementFilter(tstType);
        int foundNum = collector.WherePasses(filter).ToElementIds( ).Count;
        Trace.WriteLine(String.Format(" {0}: elements amount {1}", tstType.GetType( ).Name,
        foundNum));
    }
}
```

注意:CurveElementFilter 也可以由参数 inverted 来匹配非传入线元素类型的线型元素。

3.3.6 元素逻辑过滤器(ElementLogicalFilter)

Revit® API 有两种逻辑过滤器:LogicalOrFilter 和 LogicalAndFilter。下面介绍它们的用法。

1) LogicalOrFilter

LogicalOrFilter 是过滤器的逻辑或集合,可以查找匹配符合任意一个过滤器条件的元素。代码片段 3-46 演示了两种情形。

① 情形 1 构建了墙和标高的类别过滤器,然后将这两种过滤器通过逻辑或组合成一个逻辑或过滤器,最后将文档中符合墙和标高类别之一的所有元素查找出来。

② 情形 2 构建了墙、标高、楼板、钢筋和 MEP 系统的类过滤器集合,然后将这个集合组合成一个逻辑或过滤器,最后将文档中符合这些条件之一的元素都查找出来。

代码片段 3-46:元素过滤器

```
/// < summary>
///使用 LogicalOrFilter 过滤元素
/// < /summary>
void TestLogicalOrFilter(Document doc)
{
    //情形 1:合并两个过滤器→找到所有属于墙类别或者属于标高类别的元素
    ElementCategoryFilter filterWall =  new ElementCategoryFilter(BuiltInCategory.OST_
    Walls);
    ElementCategoryFilter filterLevel =  new ElementCategoryFilter(BuiltInCategory.OST_
    Levels);
    LogicalOrFilter orFilter =  new LogicalOrFilter(filterWall, filterLevel);
    FilteredElementCollector collector =  new FilteredElementCollector(doc);
    ICollection< Element>  founds =  collector.WherePasses(orFilter).ToElements( );
    foreach(Element elem in founds)
    {
        Trace.WriteLine(String.Format("Element Id {0}, type {1}", elem.Id.IntegerValue,
        elem.GetType( )));
    }

    //情形 2:合并两个过滤器集合→找到所有属于传入类型的元素
    Type[] elemTypes =  { typeof(Wall), typeof(Level), typeof(Floor), typeof(Rebar), typeof
    (MEPSystem)};
    List< ElementFilter>  filterSet =  new List< ElementFilter> ( );
    foreach (Type elemType in elemTypes)
    {
        ElementClassFilter filter =  new ElementClassFilter(elemType);
        filterSet.Add(filter);
    }
    orFilter =  new LogicalOrFilter(filterSet);
    collector =  new FilteredElementCollector(doc);
    founds =  collector.WherePasses(orFilter).ToElements( );
    foreach (Element elem in founds)
    {
        Trace.WriteLine(String.Format("Element Id {0}, type {1}", elem.Id.IntegerValue,
        elem.GetType( ).Name));
    }
}
```

2) LogicalAndFilter

LogicalAndFilter 是过滤器的逻辑与集合,可以查找匹配符合所有过滤器条件的元素。代码片段 3-47 演示了两种情形。

① 情形 1 构建了墙和设计选型的类过滤器,然后将这两种过滤器通过逻辑与组合成一个逻辑与过滤器,最后将文档中符合某种设计选项的对应的墙查找出来。

② 情形 2 构建了使用了情形 1 的两个类过滤器,并构建了第三个过滤器结构墙用法过滤器(StructuralWallUsageFilter),然后将这三个过滤器集合组合成一个逻辑与过滤器,最后将文档中符合某种设计选项的结构承重墙都查找出来。

代码片段 3-47:元素过滤器

```
/// < summary>
///使用 LogicalAndFilter 过滤器
/// < /summary>
void TestLogicalAndFilter(Document doc)
{
    //情形 1:合并两个过滤器->找到所有符合特定设计选项的墙
    ElementClassFilter wallFilter = new ElementClassFilter(typeof(Wall));
    FilteredElementCollector collector = new FilteredElementCollector(doc);
    ICollection< ElementId> designOptIds = collector.OfClass(typeof(DesignOption)).
    ToElementIds( );
    foreach(ElementId curId in designOptIds)
    {
        ElementDesignOptionFilter designFilter = new ElementDesignOptionFilter(curId);
        LogicalAndFilter andFilter = new LogicalAndFilter(wallFilter, designFilter);
        collector = new FilteredElementCollector(doc);
        int wallCount = collector.WherePasses(andFilter).ToElementIds( ).Count;
        Trace.WriteLine(String.Format("{0} Walls belong to DesignOption {1}.", wallCount,
        curId.IntegerValue));
    }

    //情形 2: 找到所有符合特定设计选项并且其 StructuralWallUsage 属于承重的墙
    foreach (ElementId curId in designOptIds)
    {
        //构造逻辑与过滤器
        List< ElementFilter> filters = new List< ElementFilter> ( );
        filters.Add(wallFilter);
        filters.Add(new ElementDesignOptionFilter(curId));
        filters.Add(new StructuralWallUsageFilter(StructuralWallUsage.Bearing));
        LogicalAndFilter andFilter = new LogicalAndFilter(filters);

        //应用该过滤器并遍历获取到的元素
        collector = new FilteredElementCollector(doc);
        int wallCount = collector.WherePasses(andFilter).ToElementIds( ).Count;
        Trace.WriteLine(String.Format("{0} Bearing Walls belong to DesignOption {1}.",
        wallCount, curId.IntegerValue));
    }
}
```

示例中的情形 1 也可以使用快捷方法达到同样的功能，如代码片段 3-48 所示，它将会减少代码的行数。

代码片段 3-48：元素过滤器

```
FilteredElementCollector collector =  new FilteredElementCollector(document);
//找到所有符合某种特定设计选项的墙
optionICollection< ElementId>  walls =  collector.OfClass(typeof(Wall)).
ContainedInDesignOption(myDesignOptionId).ToElementIds( );
```

第 4 章　建筑建模

本章介绍建筑建模相关的一些 API。

4.1　标高和轴网(Level/Grid)

标高和轴网相当于地球的经度和纬度，在 Revit® 里面起到“定海神针”的作用，整个建筑都是基于它们建立起来的。

4.1.1　标高

标高是有限水平平面，大多数元素都是基于标高来定位，如墙、地板、天花板、屋顶等。对应的类是 Level，它继承自 Element 类。类图见图 4-1。

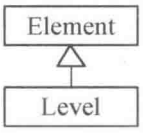


图 4-1　Level 类图

1) 标高的高度(Elevation and ProjectElevation)

通过 Level. Elevation 属性(对应的 BuiltInParameter 是 LEVEL_ELEV)可以获取标高的高度，它代表的是该标高相对于“基面(Elevation Base)”的高度，基面可以是“项目基点”，也可以是“测量点”，在 Revit® 界面上，用户可以通过标高对应的标高的类型(LevelType)属性来确定基面。

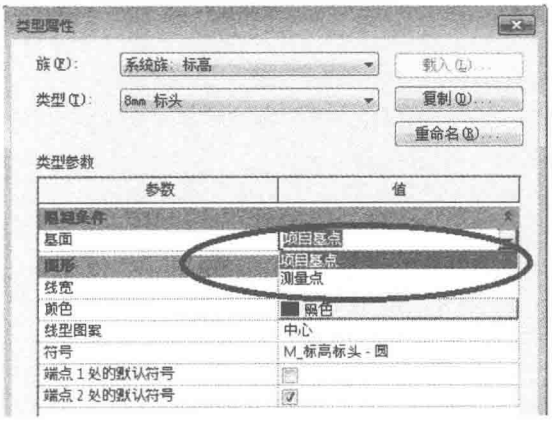


图 4-2　设置标高基面

注意：该操作将修改所有这一类型的标高的基面，如图 4-2 所示。

代码片段 4-1 展示如何用 API 来修改标高的基面，首先得到标高的标高类型，然后拿到“基面”参数，最后设置需要的值，0 代表项目基点，1 代表测量点。

代码片段 4-1：修改标高的基面

```
LevelType levelType =  RevitDoc.GetElement(level.GetTypeId( )) as LevelType;
Parameter relativeBaseType =  levelType.get_Parameter(BuiltInParameter.LEVEL_RELATIVE_
    BASE_TYPE);
relativeBaseType.Set(1); //项目基点 =  0, 测量点 =  1
```

而 Level. ProjectElevation 属性代表标高相对于项目基点的高度，等同于当基面是项目基点时 Level. Elevation 的值。

2) 创建标高

在 Revit® 界面上，用户可以在立面视图或者剖面视图上来创建标高。API 中，用户可以使用 NewLevel 方法来创建：Level Document. Create. NewLevel(double elevation)。

代码片段 4-2 是一个简单的创建标高的例子。

代码片段 4-2:创建标高

```
using (Transaction transaction = new Transaction(RevitDoc))
{
    transaction.Start("Create Level");
    Level level = RevitDoc.Create.NewLevel(10.0);
    transaction.Commit( );
}
```

注意:使用 API 创建标高之后,Revit® 不会自动创建对应的视图,这点有别于 Revit® 界面,如果想要创建对应的视图,可以使用 ViewPlan. Create 函数:ViewPlan. Create (document, viewFamilyTypeId, levelId),其中 viewFamilyTypeId 代表一个 ViewFamilyType 的 ID。

代码片段 4-3 展示如何找出所有属于 FloorPlan 或者 CeilingPlan 的视图类型,然后用这些视图类型分别创建一个视图,基于一个已有的标高。

代码片段 4-3:创建标高对应的视图

```
Level level; //已知的标高
//过滤出所有的 ViewFamilyType
var classFilter = new ElementClassFilter(typeof(ViewFamilyType));
FilteredElementCollector filteredElements = new FilteredElementCollector(RevitDoc);
filteredElements = filteredElements.WherePasses(classFilter);
foreach (ViewFamilyType viewFamilyType in filteredElements)
{
    //找到 ViewFamily 类型是 FloorPlan 或者 CeilingPlan 的 ViewFamilyType
    if (viewFamilyType.ViewFamily = = ViewFamily.FloorPlan ||
       viewFamilyType.ViewFamily = = ViewFamily.CeilingPlan)
    {
        transaction.Start("Create view of type " + viewFamilyType.ViewFamily);
        //创建视图
        ViewPlan view = ViewPlan.Create(RevitDoc, viewFamilyType.Id, level.Id);
        transaction.Commit( );
    }
}
```

4.1.2 轴网

轴网对应的类是 Grid,也继承自 Element,见图 4-3。

1) 轴网曲线(Curve)

通过 Grid. Curve 属性能够拿到轴网的曲线,如果 Grid. IsCurved 返回 true,那么 Curve 将是一个弧形曲线 Arc 对象,否则就是 Line 对象。

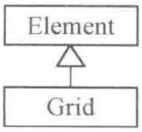


图 4-3 Grid 类图

2）创建轴网

创建轴网的函数有两个重载，分别对应于直线和弧线轴网：

- Document. Create. NewGrid(Arcarc)
- Document. Create. NewGrid(Lineline)

代码片段 4-4 展示如何创建一条直线轴网。

代码片段 4-4：创建轴网

```
using (Transaction transaction =  new Transaction(RevitDoc))
{
    transaction.Start("Create Grid");
    Grid grid =  RevitDoc.Create.NewGrid(
        Line.CreateBound(new XYZ(0, 0, 0), new XYZ(10, 10, 0)));
    grid.Name =  "BB";
    transaction.Commit( );
}
```

注意：

① 传入参数对应的直线或弧线所在的平面必须是一个水平的。

② 创建的轴网将被自动按照上一次的规则递增命名，可以是数字或者字母，取决于上一次命名的情况。例如，上一次创建的轴网名字为 A，本次就自动为 B；上一次为 3，本次则为 4；如果字母或者数字被占用，则跳过。

③ 可以通过 Grid. Name 属性来设置轴网的名称。如果被设置的名字被占用，会有 ArgumentException 抛出。

4.2　宿主元素（HostObject）

宿主元素是指可以被某些元素附着的元素，比如墙，上面可以放置门窗。天花板、地板、屋顶等，都属于宿主元素。

4.2.1　类图

宿主元素类图见图 4-4。

宿主元素类型类图见图 4-5。

图 4-4 和图 4-5 是主要宿主元素类的类图。不在图中的类，读者可以自行查阅 API 帮助文档或者使用反射工具查看。

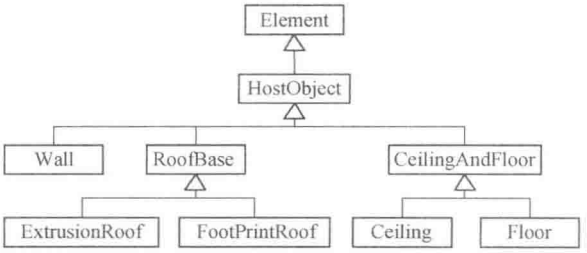


图 4-4　宿主元素类图

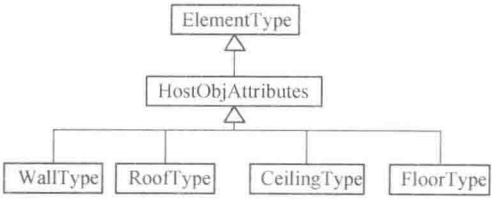


图 4-5　宿主元素类型类图

4.2.2 获取复合结构(CompoundStructure)

用户可以使用 HostObjAttributes. GetCompoundStructure()这个方法来获取复合结构 CompoundStructure。其中 HostObjAttributes 的子类包括有：WallType，FloorType 和 RoofType。

而 CompoundStructure. GetLayers()方法可以用来获取 CompoundStructureLayer，也就是每一层的信息，从而进一步获取每层的材质、厚度、功能等信息。

代码片段 4-5 展示如何获取墙的每层的厚度和材质。

代码片段 4-5：获取墙每层厚度和材质

```
Wall wall = GetElement(RevitDoc, new ElementId(732980)) as Wall;
CompoundStructure compoundStructure = wall.WallType.GetCompoundStructure( );
if (compoundStructure = = null)
    return;
if (compoundStructure.LayerCount > 0)
{
    foreach (CompoundStructureLayer compoundStructureLayer in compoundStructure.
    GetLayers( ))
    {
        //获取材质和厚度
        ElementId materialId = compoundStructureLayer.MaterialId;
        double layerWidth = compoundStructureLayer.Width;
    }
}
```

4.2.3 获取宿主上的元素

使用 HostObject. FindInserts 方法可以用来获取插入到宿主元素上的元素的集合：

IList<ElementId> FindInserts(bool addRectOpenings, bool includeShadows, bool includeEmbeddedWalls, bool includeSharedEmbeddedInserts)

参数说明如表 4-1 所示。

表 4-1　　参数说明

参数	类型	说明
addRectOpenings	Bool	设为 True，则包含洞口
includeShadows	Bool	设为 True，则包含阴影
includeEmbeddedWalls	Bool	设为 True，则包含嵌入的墙
includeSharedEmbeddedInserts	Bool	设为 True，则包含共享的嵌入墙

4.2.4 获取宿主的面

如果要在宿主上面放置构件，就需要获取宿主的面。用户除了从元素的几何信息分析出它的面，还可以使用 HostObjectUtils 提供的几个方法，更方便地获取宿主的面。方法如表 4-2 所示。

表 4-2　　获取宿主的面的方法

方法	用途
GetBottomFaces(HostObject)	获得宿主底部面的引用(Reference)
GetSideFaces(HostObject, ShellLayerType)	获得宿主侧边面的引用(Reference)
GetTopFaces(HostObject)	获得宿主顶部面的引用(Reference)

代码片段 4-6 展示如何获取一个楼板的朝上的面。

代码片段 4-6:获取楼板的上表面

```
Floor floor = GetElement< Floor> (185601);

//获取一个楼板面的引用
IList< Reference> references = HostObjectUtils.GetTopFaces(floor);
if (references.Count = = 1)
{
    var reference = references[0];

    //从引用获取面的几何对象,这里是一个 PlanarFace
    GeometryObject topFaceGeo = floor.GetGeometryObjectFromReference(reference);
    //转型成我们想要的对象
    PlanarFace topFace = topFaceGeo as PlanarFace;
}
```

4.2.5　创建墙(Wall)

创建墙的方法有五个重载。方法和用途见表 4-3。

表 4-3　　创建墙的方法和用途

方法	用途
Create(Document, Curve curve, levelId, bool structural)	给定曲线和标高,使用默认墙类型创建墙
Create(Document, IList<Curve> profile, bool structural)	给定墙的曲线形状,使用默认墙类型创建墙
Create(Document, IList < Curve > profile, ElementId wallTypeId, levelId, bool structural)	此方法和上面类似,区别是可以指定标高和墙类型,而不是使用默认的墙类型
Create(Document, IList < Curve > profile, ElementId wallTypeId, levelId, bool structural, XYZ normal)	此方法同上,区别是最后一个参数 normal 可以指定墙的朝向
Create(Document, Curve curve, ElementId wallTypeId, levelId, double height, double offset, bool flip, bool structural)	创建墙,并设置墙的高度(height 参数),底部偏移(offset 参数),是否翻转(flip)

下面对五个方法进行一一介绍。

① Wall Create(Document document, Curve curve, ElementId levelId, bool structural):代码片段 4-7 展示如何使用默认墙类型创建墙,基于一个给定的标高和一条曲线。

代码片段 4-7:创建默认墙

```
ElementId levelId = new ElementId(311);
using (Transaction transaction = new Transaction(RevitDoc))
{
    transaction.Start("Create wall");
    Wall wall = Wall.Create(RevitDoc, Line.CreateBound(new XYZ(0, 0, 0), new XYZ(100, 0,
    0)), levelId, false);
    transaction.Commit( );
}
```

② Wall Create(Document document, IList<Curve> profile, bool structural):给定墙的曲线形状,使用默认墙类型创建墙,则可以创建一个梯形的墙,实例见代码片段 4-8。

代码片段 4-8:创建梯形墙

```
IList< Curve> curves = new List< Curve> ( );
curves.Add(Line.CreateBound(new XYZ(100, 20, 0), new XYZ(100, -20, 0)));
curves.Add(Line.CreateBound(new XYZ(100, -20, 0), new XYZ(100, -10, 10)));
curves.Add(Line.CreateBound(new XYZ(100, -10, 10), new XYZ(100, 10, 10)));
curves.Add(Line.CreateBound(new XYZ(100, 10, 5), new XYZ(100, 20, 0)));
using (Transaction transaction = new Transaction(RevitDoc))
{
    transaction.Start("Create wall");
    Wall wall = Wall.Create(RevitDoc, curves, false);
    transaction.Commit( );
}
```

结果如图 4-6 所示。

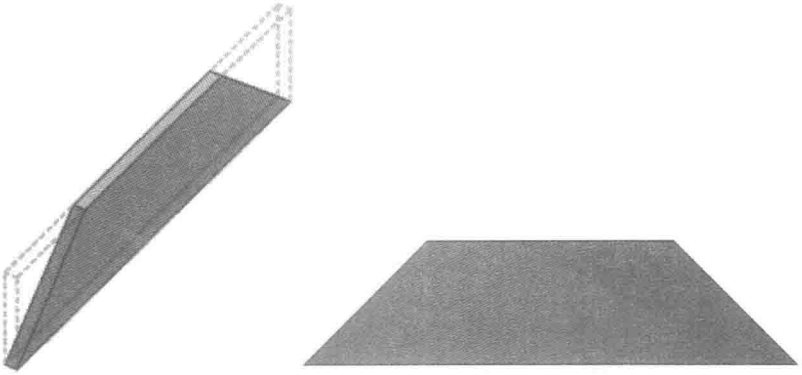


图 4-6 创建墙结果图

注意:

- 曲线形状是墙的正面,即创建的是一面墙,而不是创建一系列的墙。
- 形状是必须闭合的,否则抛异常,创建失败。
- 经编者测试,无法创建斜的墙。

③ Wall Create(Document document, IList<Curve> profile, ElementId wallTypeId,

ElementId levelId, bool structural)：此方法和上述方法类似，区别是可以指定标高和墙类型，而不是使用默认的墙类型。这里不再赘述。

④ Wall Create(Document document, IList<Curve> profile, ElementId wallTypeId, ElementId levelId, bool structural, XYZ normal)：此方法同上，区别是最后一个参数 normal 可以指定墙的朝向。如果创建的墙的朝向不符合要求，可以使用 Wall. Filp()方法来调整墙的朝向。

代码片段 4-9 展示如何使用该方法创建正反两面墙。

代码片段 4-9：创建正反两面墙

```
ElementId levelId = new ElementId(311);
ElementId wallTypeId = new ElementId(397);
IList< Curve> curves = new List< Curve> ( );

//创建第一面墙
XYZ[] vertexes = new XYZ[] { new XYZ(0, 0, 0), new XYZ(0, 100, 0), new XYZ(0, 0, 100) };
for (int ii = 0; ii < vertexes.Length; ii+ + )
{
    if (ii ! = vertexes.Length - 1)
        curves.Add(Line.CreateBound(vertexes[ii], vertexes[ii + 1]));
    else
        curves.Add(Line.CreateBound(vertexes[ii], vertexes[0]));
}
Wall wall = null;
using (Transaction transaction = new Transaction(RevitDoc))
{
    transaction.Start("Create wall 1");
    wall = Wall.Create(RevitDoc, curves, wallTypeId, levelId, false, new XYZ(-1, 0, 0));
    transaction.Commit( );
}

//创建第二面墙，面朝向相反
curves.Clear( );
vertexes = new XYZ[] { new XYZ(0, 0, 100), new XYZ(0, 100, 100), new XYZ(0, 100, 0) };
for (int ii = 0; ii < vertexes.Length; ii+ + )
{
    if (ii ! = vertexes.Length - 1)
        curves.Add(Line.CreateBound(vertexes[ii], vertexes[ii + 1]));
    else
        curves.Add(Line.CreateBound(vertexes[ii], vertexes[0]));
}
using (Transaction transaction = new Transaction(RevitDoc))
{
    transaction.Start("Create wall 2");
    wall = Wall.Create(RevitDoc, curves, wallTypeId, levelId, false, new XYZ(1, 0, 0));
    transaction.Commit( );
}
```

执行结果如图 4-7 所示。

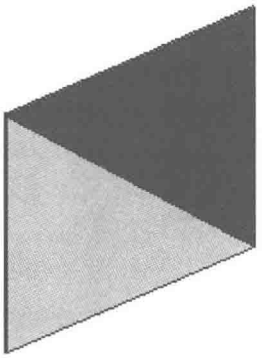


图 4-7　创建墙结果图

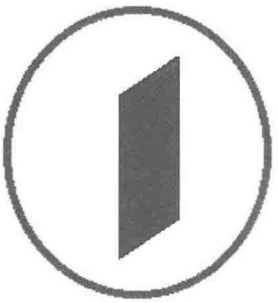
注意:这里的最后一个参数 normal(法向量)必须和墙面垂直,否则抛出异常。

⑤ Wall Create(Document document, Curve curve, ElementId wallTypeId, ElementId levelId, double height, double offset, bool flip, bool structural):创建墙,并设置墙的高度(height 参数)、底部偏移(offset 参数),是否翻转(flip)。代码片段 4-10 展示如何使用该方法创建墙。

代码片段 4-10:创建墙,并设置高度、偏移和是否翻转

```
ElementId levelId =  new ElementId(311);
ElementId wallTypeId =  new ElementId(397);

using (Transaction transaction =  new Transaction(RevitDoc))
{
    transaction.Start("Create wall");
    Wall wall =  Wall.Create(RevitDoc, Line.CreateBound(new XYZ
    (0, 0, 0), new XYZ(0, 100, 0)), wallTypeId, levelId, 200, 300,
    true, false);
    transaction.Commit( );
}
```



将生成图 4-8 中圆圈中的墙。圆圈下面的墙为上一段代码所创建,仅作参考。



图 4-8　创建墙结果图

4.2.6　创建楼板(Floor)

本节将介绍如何创建楼板。创建楼板的方法有四个重载,前面三个是创建普通楼板,最后一个是创建结构基础楼板,见表4-4。

表 4-4　　**创建楼板的方法和用途**

方法	用途
NewFloor(CurveArray profile, bool structural)	给定楼板形状,创建楼板,标高和楼板类型都是默认类型
NewFloor(CurveArray profile, floorType, level, bool structural)	给定楼板形状、类型和标高,创建楼板
NewFloor(CurveArray profile, floorType, level, bool structural, XYZ normal)	给定楼板形状、类型、标高和楼板面法向方向
NewFoundationSlab(CurveArray profile, floorType, level, bool structural, XYZ normal)	同上,区别是创建结构基础楼板

下面对四个方法进行一一介绍。

① Floor NewFloor(CurveArray profile, bool structural):本方法最简单,创建的楼板的类型是默认类型,标高是当前视图的默认标高,structrual 如果是 True,则创建结构楼板,

参见代码片段 4-11，创建一个三角形的楼板。

代码片段 4-11：创建三角形墙

```
CurveArray curveArray =  new CurveArray( );
curveArray.Append(Line.CreateBound(new XYZ(0, 0, 0), new XYZ(100, 0, 0)));
curveArray.Append(Line.CreateBound(new XYZ(100, 0, 0), new XYZ(0, 100, 0)));
curveArray.Append(Line.CreateBound(new XYZ(0, 100, 0), new XYZ(0, 0, 0)));
using (Transaction transaction =  new Transaction(RevitDoc))
{
    transaction.Start("Create floor");
    Floor floor =  RevitDoc.Create.NewFloor(curveArray, false);
    transaction.Commit( );
}
```

结果如图 4-9 所示。

② Floor NewFloor(CurveArray profile, FloorType floorType, Level level, bool structural)：本方法与上述方法基本一致，可以传入楼板类型和标高。

③ Floor NewFloor(CurveArray profile, FloorType floorType, Level level, bool structural, XYZ normal)：本方法与上一个方法的区别是多了一个 normal 用来控制楼板的面朝向。但是测试过程中发现其实没有什么效果。

④ Floor NewFoundationSlab(CurveArray profile, FloorType floorType, Level level, bool structural, XYZ normal)：本方法用来创建结构基础，使用方法与上面一个相同，这里不再赘述。创建结构基础在界面上的菜单如图 4-10 所示。

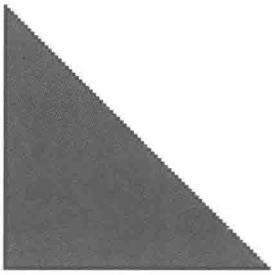


图 4-9　创建楼板结果图

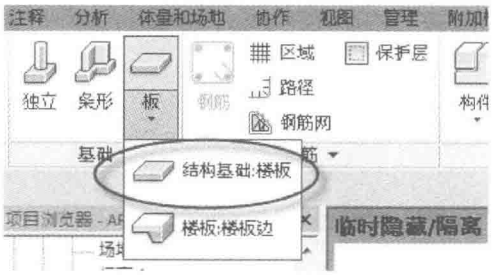


图 4-10　结构基础界面

4.2.7　天花板和屋顶(Ceiling/Roof)

天花板和屋顶类图如图 4-11 所示。

1) 创建屋顶

创建屋顶有两个重载：

- ExtrusionRoof NewExtrusionRoof (CurveArray profile, ReferencePlane refPlane, Level level, RoofType roofType, double extrusionStart, double extrusionEnd)
- FootPrintRoof NewFootPrintRoof (CurveArray

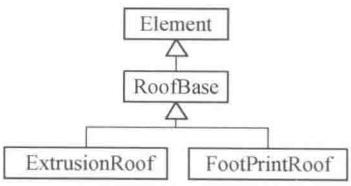


图 4-11　天花板屋顶类图

footPrint, Level level, RoofType roofType, out ModelCurveArray footPrintToModelCurvesMapping)

下面对这两个方法进行一一介绍。

① ExtrusionRoof NewExtrusionRoof(CurveArray profile, ReferencePlane refPlane, Level level, RoofType roofType, double extrusionStart, double extrusionEnd):该方法用来创建拉升屋顶,方法参数解释如表 4-5 所示。

表 4-5　　创建拉升屋顶的方法参数

参数	含义
profile	屋顶的侧面形状,注意是非封闭曲线
refPlane	参照平面
level	屋顶的标高
roofType	屋顶类型
extrusionStart	拉升起点,相对于屋顶侧面的位移,负值代表反向拉升
extrusionEnd	拉升终点,相对于屋顶侧面的位移,负值代表反向拉升

代码片段 4-12 展示如何使用该方法创建屋顶。

代码片段 4-12:创建屋顶

```
using (Transaction transaction = new Transaction(RevitDoc))
{
    View view = RevitDoc.ActiveView;
    //先创建一个参照平面
    XYZ bubbleEnd = new XYZ(0, 0, 0);
    XYZ freeEnd = new XYZ(0, 100, 0);
    XYZ thirdPnt = new XYZ(0, 0, 100);
    transaction.Start("Create reference plane");
     ReferencePlane plane = RevitDoc.Create.NewReferencePlane2(bubbleEnd, freeEnd,
     thirdPnt, view);
    transaction.Commit( );
    //创建屋顶前准备参数
    Level level = RevitDoc.GetElement(new ElementId(311)) as Level;
    RoofType roofType = RevitDoc.GetElement(new ElementId(335)) as RoofType;
    CurveArray curveArray = new CurveArray( );
    curveArray.Append(Line.CreateBound(new XYZ(0, 0, 50), new XYZ(0, 50, 100)));
    curveArray.Append(Line.CreateBound(new XYZ(0, 50, 100), new XYZ(0, 100, 50)));
    //创建屋顶
    transaction.Start("Create roof");
    RevitDoc.Create.NewExtrusionRoof(curveArray, plane, level, roofType, 10, 200);
    transaction.Commit( );
}
```

运行结果如图 4-12 所示。

② FootPrintRoof NewFootPrintRoof(CurveArray footPrint, Level level, RoofType roofType, out ModelCurveArray footPrintToModelCurvesMapping):本方法用来创建迹线屋顶,函数参数含义见表 4-6。

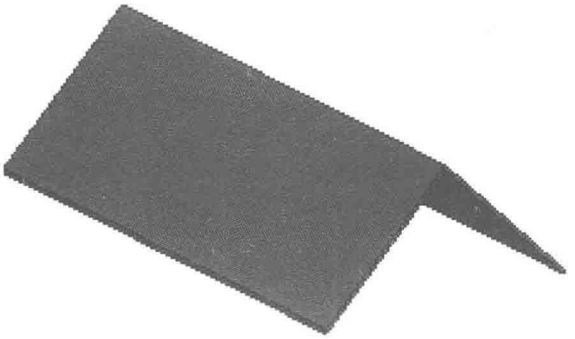


图 4-12　创建屋顶结果图

表 4-6　　创建迹线屋顶的方法参数

参数	含义
footPrint	屋顶的形状，一个或多个闭合环线
level	屋顶的标高
roofType	屋顶类型
footPrintToModelCurvesMapping	获取输入屋顶曲线对应的模型线，模型线可以进一步用来对屋顶进行控制，比如通过模型线设置对应的屋顶坡度或者偏移

代码片段 4-13 创建了一个屋顶，并设置使用获得的模型线设置屋顶的坡度。

代码片段 4-13：创建带洞口屋顶

```
using (Transaction transaction =  new Transaction(RevitDoc))
{
    //创建屋顶前准备参数
    Level level =  RevitDoc.GetElement(new ElementId(311)) as Level;
    RoofType roofType =  RevitDoc.GetElement(new ElementId(335)) as RoofType;
    CurveArray curveArray =  new CurveArray( );
    //屋顶外边框
    curveArray.Append(Line.CreateBound(new XYZ(0, 0, 0), new XYZ(30, 0, 0)));
    curveArray.Append(Line.CreateBound(new XYZ(30, 0, 0), new XYZ(30, 30, 0)));
    curveArray.Append(Line.CreateBound(new XYZ(30, 30, 0), new XYZ(0, 30, 0)));
    curveArray.Append(Line.CreateBound(new XYZ(0, 30, 0), new XYZ(0, 0, 0)));
    //在中间添加洞口
    curveArray.Append(Line.CreateBound(new XYZ(5, 5, 0), new XYZ(5, 15, 0)));
    curveArray.Append(Line.CreateBound(new XYZ(5, 15, 0), new XYZ(15, 5, 0)));
    curveArray.Append(Line.CreateBound(new XYZ(15, 5, 0), new XYZ(5, 5, 0)));

    //创建屋顶
    transaction.Start("Create roof");
    ModelCurveArray modelCurveArray =  new ModelCurveArray( );
     FootPrintRoof roof =  RevitDoc.Create.NewFootPrintRoof(curveArray, level, roofType,
     out modelCurveArray);
    //设置屋顶坡度
    ModelCurve curve1 =  modelCurveArray.get_Item(0);
```

```
    ModelCurve curve3 = modelCurveArray.get_Item(2);
    roof.set_DefinesSlope(curve1, true);
    roof.set_SlopeAngle(curve1, 0.5);
    roof.set_DefinesSlope(curve3, true);
    roof.set_SlopeAngle(curve3, 1.6);
    transaction.Commit( );
}
```

运行结果如图 4-13 所示。

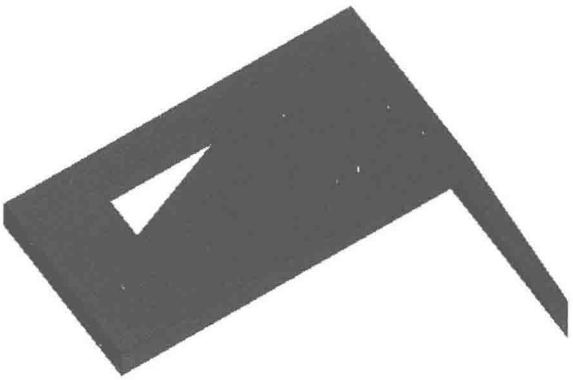


图 4-13　创建屋顶结果图

2）获取屋顶数据

拉升屋顶一个重要的方法是 GetProfile()返回 ModelCurveArray 的对象，用来获取屋顶的侧面形状，与创建拉升屋顶时传入的 CurveArray 的对象一一对应。

对于迹线屋顶，可以使用最后一个 out 参数 footPrintToModelCurvesMapping 获取的 ModelCurve 来判断屋顶是否有坡度，以及坡度大小。

3）创建天花板

目前尚没有 API 支持创建天花板。

4.3　族实例（FamilyInstance）

族实例是通过族的类型创建的实例。

4.3.1　族实例的宿主（Host）

通过 FamilyInstance. Host 方法获取族实例的宿主。

4.3.2　旋转实例（Rotation）

FamilyInstance. rotate()，旋转 180°。

4.3.3　门和窗（Door/Window）

门窗除了有 Room 属性，还有 FromRoom 和 ToRoom 属性，用来获取门窗和房间的关系。

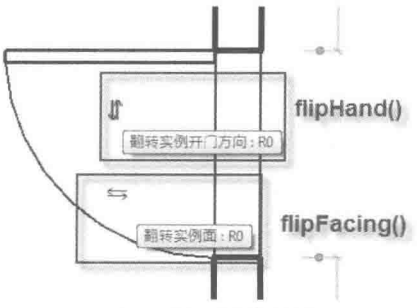


图 4-14　翻转门窗

4.3.4　翻转门窗（Flip）

使用 FamilyInstance. flipFacing()翻转门窗的朝向，而 CanFlipFacing 属性可以用来判断能否翻转，FacingFlipped 属性判断是否被翻转过。

同理，使用 FamilyInstance. flipHand()翻转开门方向，而 CanFlipHand 属性判断能否翻转开门方向，HandFlipped 属性返回是否被翻转过。

示例图见图 4-14。

4.3.5　子构件和父构件（SubComponent/SuperComponent）

使用 FamilyInstance. GetSubComponentIds()可以获取族实例子构件，使用 Component. SuperComponent 可以直接获取构件的父构件。见图 4-15。

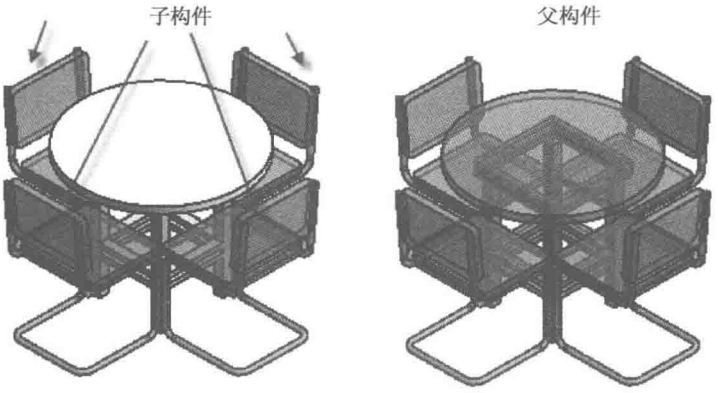


图 4-15　子构件和父构件

4.4　族实例(FamilyInstance)的创建

由于创建族实例比较重要，同时创建族实例的重载方法比较烦杂，于是专门列出本节来介绍一下这些方法。

创建族实例的方法名为 CreateFamilyInstance，存在于类 Autodesk. Revit. Creation. Document 以及它的父类 Autodesk. Revit. Creation. ItemFactoryBase 中。它们的对象可以从 Audodesk. Revit. DB. Document. Create 获得，下面的示例代码中的 m_creation 就是一个 Autodesk. Revit. Creation. ItemFactoryBase 或者 Autodesk. Revit. Creation. Document 对象，即：m_creation = RevitDoc. Create。

4.4.1　创建独立的族实例

NewFamilyInstance (XYZ location, FamilySymbol symbol, StructuralType structuralType)：创建一个独立的，不基于任何元素的族实例。下面的例子创建了一个家具，如图 4-16 所示。

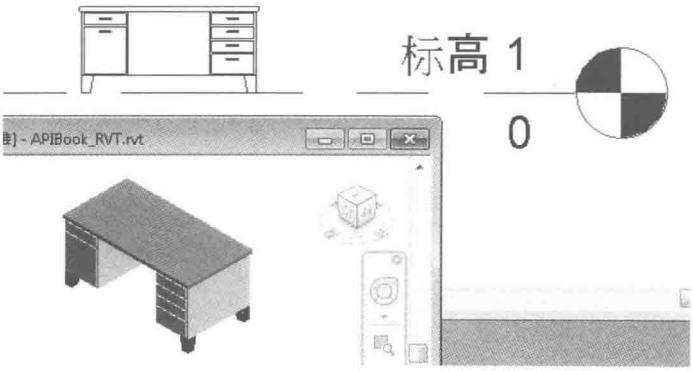


图 4-16　创建一个桌子

代码片段 4-14 展示如何使用该方法创建独立实例。

代码片段 4-14:创建独立实例

```
// place a furniture at (0, 0, 0)
FamilySymbol familySymbol = RevitDoc.GetElement(new ElementId(99774)) as FamilySymbol;
using (Transaction transaction = new Transaction(RevitDoc))
{
    transaction.Start("Create standard-alone instance");
    FamilyInstance familyInstance = m_creation.NewFamilyInstance(
        new XYZ(0, 0, 0), familySymbol, StructuralType.NonStructural);
    transaction.Commit( );
    Trace.WriteLine(familyInstance.Id);
}
```

4.4.2 创建基于宿主的族实例

创建基于宿主的族实例,有如下几个方法可供选择:

① 给定一个标高:NewFamilyInstance(XYZ location, FamilySymbol symbol, Element host, Level level, StructuralType structuralType)

② 不给定标高:NewFamilyInstance(XYZ location, FamilySymbol symbol, Element host, StructuralType structuralType)

③ 给定一个朝向:NewFamilyInstance(XYZ location, FamilySymbol symbol, XYZ referenceDirection, Element host, StructuralType structuralType)

其中 XYZ location 指定实例的位置,FamilySymbol symbol 给定实例对应的族类型,Element host 即为宿主元素,StructuralType 指定结构类型。

代码片段 4-15 创建了一个基于墙的门,如图 4-17 所示。

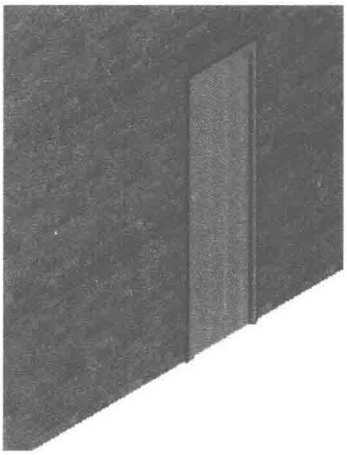


图 4-17 创建门

代码片段 4-15:墙上创建门

```
//在墙上创建一扇门
FamilySymbol familySymbol = RevitDoc.GetElement(new ElementId(49480)) as FamilySymbol;
Level level = RevitDoc.GetElement(new ElementId(30)) as Level;
Wall hostWall = RevitDoc.GetElement(new ElementId(180736)) as Wall;
using (Transaction transaction = new Transaction(RevitDoc))
{
    transaction.Start("Create standard-alone instance");
    FamilyInstance familyInstance = m_creation.NewFamilyInstance(
        new XYZ(0, 0, 0), familySymbol, hostWall, level, StructuralType.NonStructural);
    transaction.Commit( );
    Trace.WriteLine(familyInstance.Id);
}
```

注意：如果 XYZ location 如果超出了宿主本身的范围，创建可能会失败，并有错误框弹出，提示无法创建。

4.4.3　创建基于标高的族实例

NewFamilyInstance（XYZ location，FamilySymbol symbol，Level level，StructuralType structuralType）：创建一个基于标高的族实例。代码片段 4-16 创建一个基于标高的柱子，如图4-18所示。

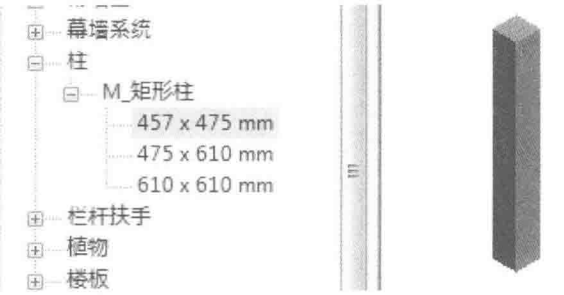


图 4-18　创建柱子结果图

代码片段 4-16：创建柱子

```
// place a column at (0, 0, 0),
FamilySymbol familySymbol =  RevitDoc.GetElement(new ElementId(52557)) as FamilySymbol;
Level level =  RevitDoc.GetElement(new ElementId(331)) as Level;
using (Transaction transaction =  new Transaction(RevitDoc))
{
    transaction.Start("Create a level based instance");
    FamilyInstance familyInstance =  m_creation.NewFamilyInstance(
        new XYZ(0, 0, 0), familySymbol, level, StructuralType.NonStructural);
    transaction.Commit( );
    Trace.WriteLine(familyInstance.Id);
}
```

4.4.4　创建基于标高的线形族实例

如果实例是线形的，就不能用一个点 XYZ 来指定其要创建的位置，而是使用 Curve，下面的方法可以用来创建一个基于标高的线形实例：

- NewFamilyInstance（Curve curve，FamilySymbol symbol，Level level，StructuralType structuralType）

代码片段 4-17 创建了一个基于标高的曲线结构梁，如图 4-19 所示。

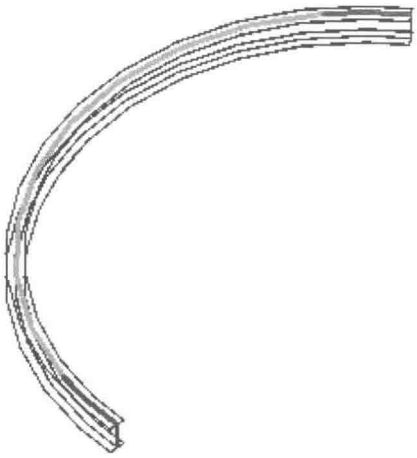


图 4-19　创建线形族实例

代码片段 4-17:创建线形实例

```
// create a beam with an arc
FamilySymbol familySymbol =  RevitDoc.GetElement(new ElementId(68912)) as FamilySymbol;
Level level =  RevitDoc.GetElement(new ElementId(339)) as Level;
using (Transaction transaction =  new Transaction(RevitDoc))
{
    transaction.Start("Create standard-alone instance");
    FamilyInstance familyInstance =  m_creation.NewFamilyInstance(
        Arc.Create(XYZ.Zero,10, 0, 2.5,new XYZ(1,0, 0),new XYZ(0, 1,0)),
        familySymbol, level, StructuralType.Beam);
    transaction.Commit( );
    Trace.WriteLine(familyInstance.Id);
}
```

注意:StructuralType 要设置成 Beam,否则创建会失败(返回值为 null)。

4.4.5 创建基于视图的二维族实例

NewFamilyInstance(XYZ origin, FamilySymbol symbol, View specView):本方法用来创建基于视图的 2D 族实例,如注释(Annotation),详细组件(detail components),标题栏(Title block)。

代码片段 4-18 在图纸中插入一个标题栏,如图 4-20 所示。

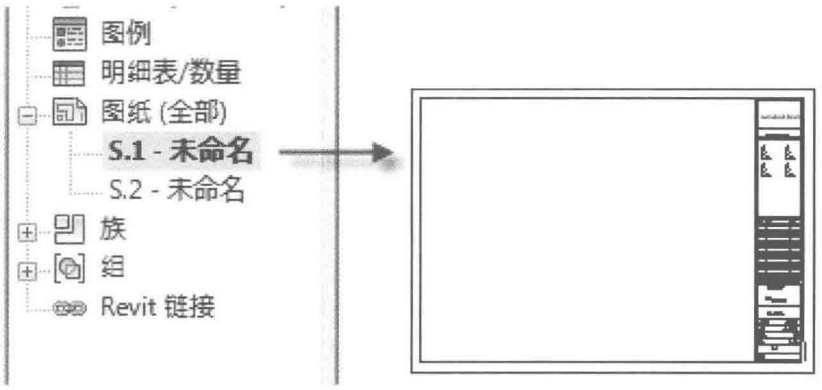


图 4-20 创建二维族实例

代码片段 4-18:创建二维实例

```
//找到标题栏族类型
var titleBlockSymbols =  new FilteredElementCollector(RevitDoc).
    OfCategory(BuiltInCategory.OST_TitleBlocks).WhereElementIsElementType( );
FamilySymbol titleBlockSymbol =  titleBlockSymbols.FirstElement( ) as FamilySymbol;

//获取一个 ViewSheet
var viewSheet =  RevitDoc.GetElement(new ElementId(175782)) as ViewSheet;

//在 ViewSheet 里面插入标题栏
```

```
using (Transaction transaction =  new Transaction(RevitDoc))
{
    transaction.Start("插入标题栏");

     var familyInstance =  m_creation.NewFamilyInstance(XYZ.Zero, titleBlockSymbol,
     viewSheet);

    transaction.Commit( );
}
```

4.4.6　创建基于视图的二维线形族实例

NewFamilyInstance(Lineline, FamilySymbol symbol, View specView)：本方法用来创建基于视图的二维线形族实例，代码片段 4-19 创建了一个基于视图的二维箭头，如图 4-21 所示。



图 4-21　创建二维线形族实例

代码片段 4-19：创建二维线形实例

```
//获取二维箭头族类型
FamilySymbol arrowSymbol =  RevitDoc.GetElement(new ElementId(176135)) as FamilySymbol;

//找到一个二维视图
var viewPlans =  new FilteredElementCollector(RevitDoc).OfClass(typeof(ViewPlan)).OfType
< ViewPlan> ( );
var viewPlan =  viewPlans.FirstOrDefault(v = >  v.GenLevel ! =  null);
var zPosition =  viewPlan.GenLevel.Elevation;
//创建族实例
using (Transaction transaction =  new Transaction(RevitDoc))
{
    transaction.Start("NewFamilyInstance");

    var familyInstance =  m_creation.NewFamilyInstance(
        Line.CreateBound(new XYZ(0, 0, zPosition), new XYZ(10, 10, zPosition)),
```

```
        arrowSymbol, viewPlan);
    Trace.WriteLine("Created family instance: " + familyInstance.Id);

    transaction.Commit( );
}
```

注意：如果传入的 line 不在 specView 上（一般是 Z 坐标没有达到 specView 对应的高度），会有异常抛出："ArgumentException: The line is not in the plane of view"。

4.4.7 创建基于面的族实例

下面两个方法用来创建基于面的族实例：

- NewFamilyInstance (Reference reference, XYZ location, XYZ referenceDirection, FamilySymbol symbol)
- NewFamilyInstance (Faceface, XYZ location, XYZ referenceDirection, FamilySymbol symbol)

代码片段 4-20 创建了一个基于墙的灯具，如图 4-22 所示。

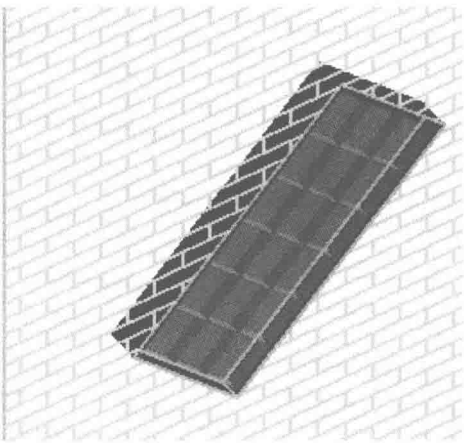


图 4-22 创建基于面的族实例

代码片段 4-20：创建基于面的族实例

```
//获取类型
FamilySymbol arrowSymbol = RevitDoc.GetElement(new ElementId(574219)) as FamilySymbol;

//找到墙的一个面
var walls = new FilteredElementCollector(RevitDoc).OfClass(typeof(Wall)).OfType< Wall>
( );
var wall = walls.FirstOrDefault( );
var faceReferences = HostObjectUtils.GetSideFaces(wall, ShellLayerType.Exterior);

//创建实例
using (Transaction transaction = new Transaction(RevitDoc))
{
    transaction.Start("NewFamilyInstance");

    var familyInstance = m_creation.NewFamilyInstance(faceReferences[0],
        new XYZ(-51.659444225, 5.702513250, 10), new XYZ(0, 1, 1), arrowSymbol);
    Trace.WriteLine("Created family instance: " + familyInstance.Id);

    transaction.Commit( );
}
```

注意：referenceDirection 必须和面平行。

4.4.8　创建基于面的线形族实例

下面两个方法用来创建基于面的线形族实例：

- NewFamilyInstance(Face face, Line position, FamilySymbol symbol)
- NewFamilyInstance(Reference reference, Line position, FamilySymbol symbol)

代码片段 4-21 创建了一个基于楼板的线性加强板，如图 4-23 所示。

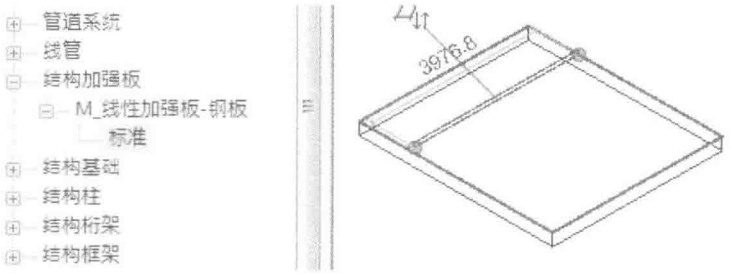


图 4-23　创建基于面的线形族实例

代码片段 4-21：创建基于面的线形实例

```
//获取类型
FamilySymbol stiffenerSymbol = RevitDoc.GetElement(new ElementId(97916)) as FamilySymbol;

//找到楼板的一个面
var floors = new FilteredElementCollector(RevitDoc).OfClass(typeof(Floor)).OfType< Floor
> ( );
var floor = floors.FirstOrDefault( );
var faceReferences = HostObjectUtils.GetTopFaces(floor);

//创建实例
using (Transaction transaction = new Transaction(RevitDoc))
{
    transaction.Start("NewFamilyInstance");

    var familyInstance = m_creation.NewFamilyInstance(
        faceReferences[0],
        Line.CreateBound(new XYZ(36, -40, 0), new XYZ(36, -20, 0)),
        stiffenerSymbol);
    Trace.WriteLine("Created family instance: " + familyInstance.Id);

    transaction.Commit( );
}
```

注意：position(Line)必须在面上，否则会有异常抛出。

4.4.9　批量创建族实例

ICollection<ElementId> NewFamilyInstances2(List<FamilyInstanceCreationData> dataList)：本方法用来批量创建族实例，以提高程序运行效率。

如果需要创建比较多的族实例，那么使用上述批量创建方法效率会比较高。使用方法

是给此方法传入一个的参数(FamilyInstanceCreationData)集合，里面每个参数封装了用来创建实例的一系列数据。

代码片段 4-22 使用批量创建函数创建了一系列的斑马线，如图 4-24 所示。

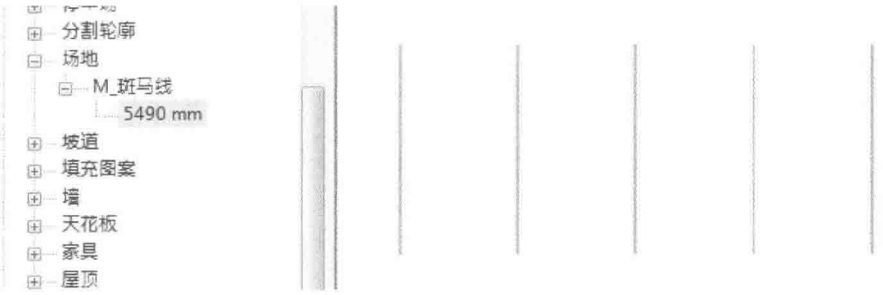


图 4-24　批量创建族实例

代码片段 4-22:批量创建人行道

```
//创建一系列的人行道

//获取类型
FamilySymbol pavementSymbol =  RevitDoc.GetElement(new ElementId(182412)) as FamilySymbol;

//准备创建数据
List< FamilyInstanceCreationData> list =  new List< FamilyInstanceCreationData> ( );
list.Add(new FamilyInstanceCreationData(new XYZ(10, 0, 0), pavementSymbol, StructuralType.
    NonStructural));
list.Add(new FamilyInstanceCreationData(new XYZ(20, 0, 0), pavementSymbol, StructuralType.
    NonStructural));
list.Add(new FamilyInstanceCreationData(new XYZ(30, 0, 0), pavementSymbol, StructuralType.
    NonStructural));
list.Add(new FamilyInstanceCreationData(new XYZ(40, 0, 0), pavementSymbol, StructuralType.
    NonStructural));
list.Add(new FamilyInstanceCreationData(new XYZ(50, 0, 0), pavementSymbol, StructuralType.
    NonStructural));

//创建实例
using (Transaction transaction =  new Transaction(RevitDoc))
{
    transaction.Start("Create pavements");
    var familyInstances =  m_creation.NewFamilyInstances2(list);
    transaction.Commit( );
    foreach (var familyInstanceId in familyInstances)
    {
        Trace.WriteLine(familyInstanceId);
    }
}
```

4.5　房间和面积(Room/Area)

4.5.1　房间、面积、标记(Room/Area/Tag)

房间(Room)，面积(Area)和空间(Space)类都继承自 SpatialElement。类图见图 4-25。

Room. Name 属性的值是由两个值拼接起来的，分别是参数“名称”和“编号”，它们对应的 BuiltInParameter 是 ROOM_NAME 和 ROOM_NUMBER。如图 4-26 所示，Room. Name 的值“Room 9”就是 ROOM_NAME“房间”和 ROOM_NUMBER“9”组成的。

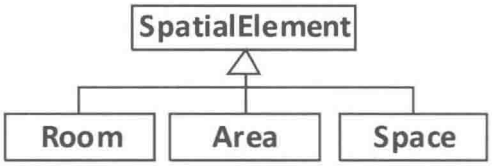


图 4-25　空间相关类的类图

房间标记 RoomTag，面积标记 AreaTag 和空间标记 SpaceTag 都继承自 SpatialElementTag 类。类图见图 4-27。

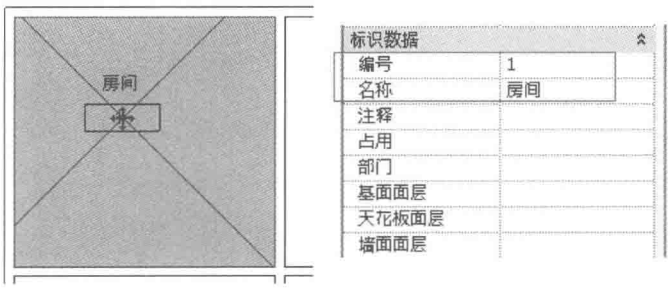


图 4-26　房间名称和编号图

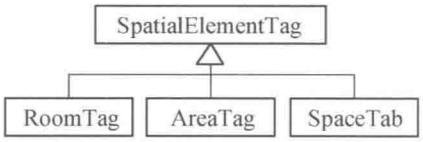


图 4-27　空间类的标签类图

RoomTag/AreaTag：这两个类属于“标记元素”(Annotation)一类，它们都是基于 View 的二维元素，通过属性 RoomTag. View 或者 AreaTag. View 能够拿到对应的 View。它们的名字都是自动计算出来的，无法通过 API 修改。

RoomTag. Room 可以获取对应的 Room，但是没有直接的方法可以从 Room 获取到对应的 RoomTag，可以先过滤出所有的 RoomTag，然后判断它们对应的 Room 属性是否是当前的 Room，如果是，则说明它们有关联关系。

4.5.2　创建房间

创建 Room 的方法是 Document. Create. NewRoom，它有很多重载，下面作简要介绍。

① Room NewRoom(Phase phase)：给定 Phase，创建一个未放置的房间。

② NewRooms2(Phase, int count)：在给定的 Phase 上创建多个房间，count 代表房间的个数。

③ Room NewRoom(Level level, UV point)：在给定标高上创建一个被放置的房间，放置点由 UV 参数指定。

- 如果在该放置点有闭合回路(Plan Circuit)，那么创建的房间会自动绑定该回路。
- 如果在该放置点没有房间边界可以绑定，会有警告框弹出：房间不在完全闭合的区域中。
- 如果在该放置点已有一个房间，也会有警告框弹出：多个房间位于同一个闭合区

域中。

④ Room NewRoom(Room room, PlanCircuit circuit):在给定的闭合回路放置房间。

- 如果参数 room 为空,则创建一个房间。
- 如果参数 room 代表的房间没有被放置,首先检查该房间的 Phase 是不是和 PlanCircuit 一致,如果一致则在指定的 PlanCircuit 放置该房间,如果不一致,就会有异常抛出。
- 如果房间已经被放置,就会有 ArgumentException 抛出。

代码片段 4-23 展示如何获取标高上的封闭区间,然后使用该封闭区间创建房间。

代码片段 4-23:创建房间

```
Level level; //此处省略获取 Level 的代码
Phase phase; //此处省略获取 Phase 的代码
//获取基于标高 level 的一个视图
var defaultView = new FilteredElementCollector(RevitDoc)
    .WherePasses(new ElementClassFilter(typeof(View)))
    .Cast< View> ( )
    .Where(v = > v.GenLevel ! = null && v.GenLevel.Id = = level.Id)
    .FirstOrDefault( );
//确保视图不为空
if (defaultView ! = null)
{
    var defaultPhase = defaultView.get_Parameter(BuiltInParameter.VIEW_PHASE);
    if (defaultPhase ! = null && defaultPhase.AsElementId( ) = = phase.Id)
    {
        using (Transaction transaction = new Transaction(RevitDoc))
        {
            transaction.Start("get_PlanTopology");
            var circuits = RevitDoc.get_PlanTopology(level, phase).Circuits;
            transaction.Commit( );
            foreach (PlanCircuit planCircuit in circuits)
            {
                RevitDoc.Create.NewRoom(null, planCircuit);
            }
        }
    }
}
```

⑤ NewRooms2(Level):给定标高,在所有可用的闭合区域创建房间,Phase 为当前默认的 Phase。等同于调用 NewRooms2(Level, theDefaultPhase),其中 theDefaultPhase 是当前默认的 Phase。

⑥ NewRooms2(Level, Phase):给定的标高和项目阶段,在所有可用的闭合区域分别创建房间。

注意:Level 的项目阶段(Level 对应的默认的视图的项目阶段)必须和给定的 Phase 保持一致,否则就会有异常抛出。

⑦ 其他一些和创建房间相关函数:ModelCurveArray NewRoomBoundaryLines

(SketchPlane sketchPlane, CurveArray curves, View view)。给定一个草图视图,一个曲线列表和一个平面视图,创建一组房间边界曲线(ModelCurveArray)。调用该方法(或多次调用)形成一个闭合区域之后,取该区域的一个点,再调用 NewRoom 就可以创建房间。

4.5.3　创建面积

创建 Area 的方法是 Document. Create. NewArea,下面是各个重载函数的介绍和一段代码实例。

① Area NewArea(ViewPlan areaView, UV point):在给定的一个面积平面视图(AreaPlan)和一个放置点(UV),在该放置点创建一个面积。如果该视图不是面积视图,会有异常 InvalidOperationException 抛出。

- 如果在该放置点有闭合回路,那么创建的面积会自动绑定该回路。
- 如果在该放置点没有房间边界可以绑定,会有警告框弹出:面积不在完全闭合的区域中。
- 如果在该放置点已有一个房间,也会有警告框弹出:多个房间位于同一个闭合区域中。

代码片段 4-24 展示如何使用该方法创建面积。

代码片段 4-24:创建面积

```
using (Transaction transaction =  new Transaction(RevitDoc))
{
    transaction.Start("Create Area");
    //如果在点(30.0, 30.0)这个位置找不到面积边界,将会有警告对话框弹出。
    Area area =  RevitDoc.Create.NewArea(areaView, new UV(30.0, 30.0));
    transaction.Commit( );
}
```

② NewAreas(List<AreaCreationData> dataList):给定一系列的输入数据,每一组数据包含一个平面视图(ViewPlan)和一个放置点(UV),来创建一系列的面积,效果等同于使用同样的输入而多次调用 NewArea。

③ 其他一些和创建面积相关函数:ModelCurve NewAreaBoundaryLine(SketchPlane sketchPlane, Curve geometryCurve, ViewPlan areaView)。给定一个草图视图,一条曲线和一个面积视图,创建一条面积边界曲线(ModelCurve)。调用该方法形成一个闭合区域之后,取该区域的一个点,再调用 NewArea 就可以在该点处创建面积了。

代码片段 4-25 创建了一个四方形闭合区域,然后在该区域内的一个点处创建一个面积。

注意:一般而言,该区域必须是闭合或者接近闭合(误差不超过 1 英寸)。

代码片段 4-25:创建四方形闭合区域并在其上创建面积

```
using (Transaction transaction =  new Transaction(RevitDoc))
{
    var create =  RevitDoc.Create;
    //通过创建四条面积边界线来形成一个正方形的闭合区域
    transaction.Start("Create Area Boundary");
    var sketchPlane =  areaView.SketchPlane;
```

```
    create.NewAreaBoundaryLine(sketchPlane,
        Line.CreateBound(new XYZ(20, 20, 0), new XYZ(40, 20, 0)), areaView);
    create.NewAreaBoundaryLine(sketchPlane,
        Line.CreateBound(new XYZ(40, 20, 0), new XYZ(40, 40, 0)), areaView);
    create.NewAreaBoundaryLine(sketchPlane,
        Line.CreateBound(new XYZ(40, 40, 0), new XYZ(20, 40, 0)), areaView);
    create.NewAreaBoundaryLine(sketchPlane,
        Line.CreateBound(new XYZ(20, 40, 0), new XYZ(20, 20, 0)), areaView);
    transaction.Commit( );

    //在新创建的面积边界的中心点(30.0, 30.0)位置放置一个面积
    transaction.Start("Create Area");
    Area area =  create.NewArea(areaView, new UV(30.0, 30.0));
    transaction.Commit( );
}
```

4.5.4 房间边界

房间的边界是一个闭合区间，房间边界可以是下面的元素：

- 墙
- 房间分隔
- 建筑柱
- 屋顶
- 楼板
- 天花板

元素可以被作为房间的边界，需满足下面的条件：

- 对于模型曲线，它的类别必须是 BuiltInCategory.OST_AreaSeparationLines。
- 对于其他元素，比如墙、柱或屋顶，它们的“房间边界”(BuiltInParameter. WALL _ ATTR _ ROOM _ BOUNDING)参数必须设为 True。见图 4-28。



图 4-28 元素的属性：房间边界

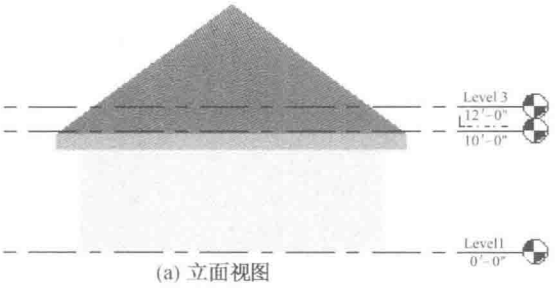
下面是一个屋顶如何作为房间边界的例子，如图 4-29 所示，房间是在标高 3 上面创建的，(a)图是立面视图，(b)图是三维视图，(c)图显示房间的边界。

面积的边界只能是模型曲线(ModelCurve)，但是房间的边界可以是墙或者其他元素，如图 4-29 中，房间边界对应的元素就是屋顶。

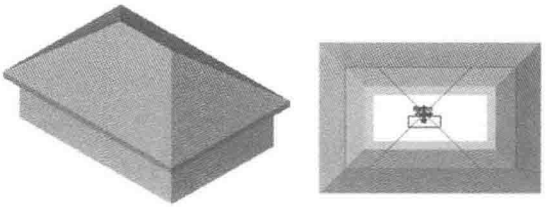
在 Revit® 界面上，选择一个房间之后，对应的边界就会高亮显示，如图 4-30 所示。

在 API 里面，用户可以调用 Room. GetBoundarySegments (SpatialElement-BoundaryOptions options)来得到房间边界，返回值是 BoundarySegment 集合的集合 IList < IList < BoundarySegment > >。SpatialElementBoundaryOptions 有两个重要属性：SpatialElementBoundaryLocation 和 StoreFreeBoundaryFaces，分别指定边界的计算方式和是否计算自由边界面。

房间边界相关的类图如图 4-31 所示。



(a) 立面视图



(b) 三维视图　　(c) 房间边界

图 4-29　创建房间

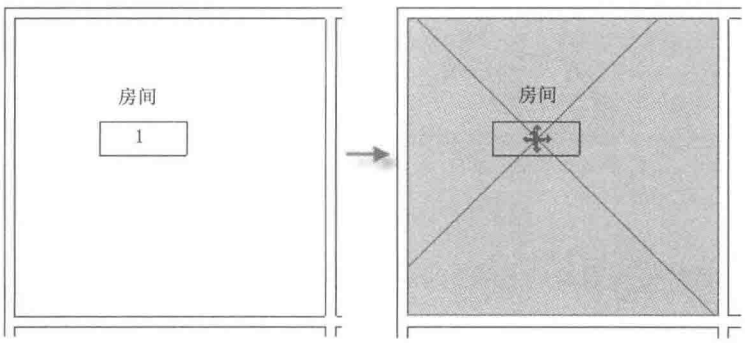


图 4-30　房间边界

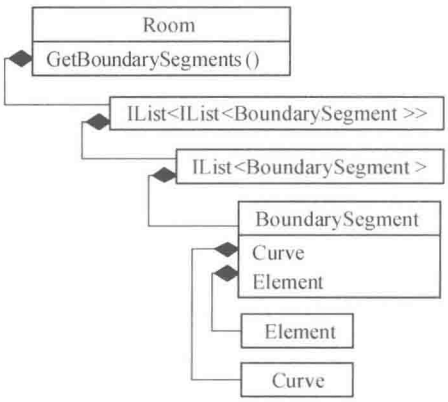


图 4-31　房间边界相关类图

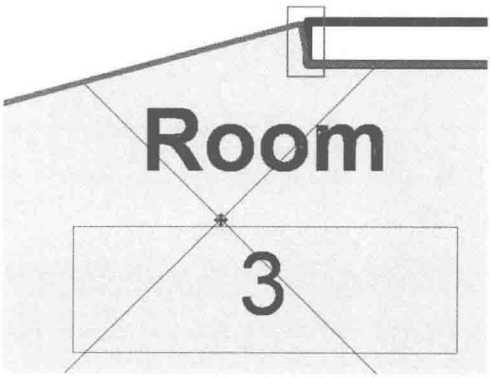


图 4-32　墙两端作为房间边界

注意：当墙的两端作为边界的时候，BoundarySegment 的 Element 属性返回值为空，但是 Curve 属性返回值不是空，如图 4-32 所示。

4.5.5 平面拓扑(PlanTopology)

在平面图上,房间依赖于拓扑结构,而拓扑结构是有一系列的元素形成的,比如墙或房间分隔,可以通过调用以下函数来获取拓扑结构:

- Document. get_PlanTopologies()
- Document. get_PlanTopologies(Phase)
- Document. get_PlanTopology(Level)
- Document. get_PlanTopology(Level, Phase)

和房间边界一样,只有房间分隔或者是那些"房间边界"是 True 的元素才能成为拓扑结构的边界。

图 4-33 是相关的类图。

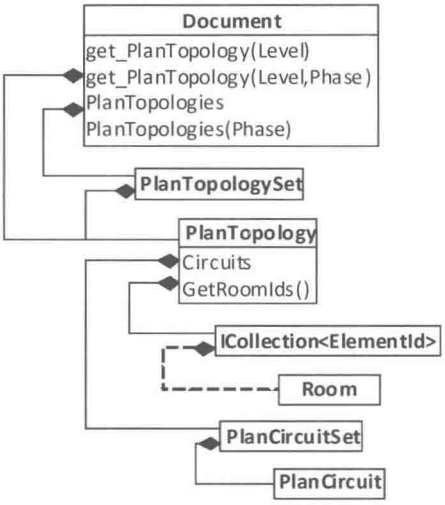


图 4-33 房间和拓扑结构图

PlanCircuit. SideNum 属性返回的是回路边数,图 4-34 列出了四种情形和它们对应的 SideNum 的值。

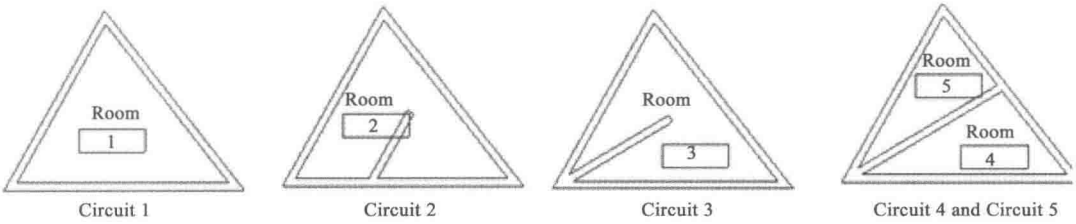


图 4-34 闭合区域的 SideNum

表 4-7 列出每种情况对应的 SideNum 的值。

表 4-7 闭合区域的 SideNum 值

Circuit	Circuit. SideNum	Circuit	Circuit. SideNum
Circuit 1	3	Circuit 4	3
Circuit 2	4 +2 = 6	Circuit 5	3
Circuit 3	3 +2 = 5		

注意:调用 Document. PlanTopologies 属性的时候,需要使用 Transaction,如代码片段 4-26 所示。

代码片段 4-26:获取拓扑结构

```
using (Transaction transaction =  new Transaction(RevitDoc))
{
    transaction.Start("GetPlanTopologies");
    PlanTopologySet planTopologies =  RevitDoc.PlanTopologies;
    transaction.Commit( );
}
```

4.5.6　房间和实例

族实例和房间的关系分两种：在房间里，如桌子、床等；介于两个房间之间，如门和窗。

族实例跟房间相关的属性有：Room，FromRoom 和 ToRoom。如果想得到跟某个项目阶段(Phase)相关的房间，对应的分别使用 get_Room(Phase)，get_FromRoom(Phase)和 get_ToRoom(Phase)这三种方法。

对应的 Space(空间)都有类似的方法，这里不赘述。

注意：

① 所有的族实例都有 Room 属性，表示在最后一个设计阶段(Phase)该实例被放置的房间。

② 只有门和窗的 FromRoom 和 ToRoom 的属性不为空，其他族实例都是空。

③ 对于门和窗，我们可以通过翻动(Flip)改变它的 Room，对于其他实例，它所处的房间边界中的房间就是它的 Room。

图 4-35 的例子是关于门和房间的关系。

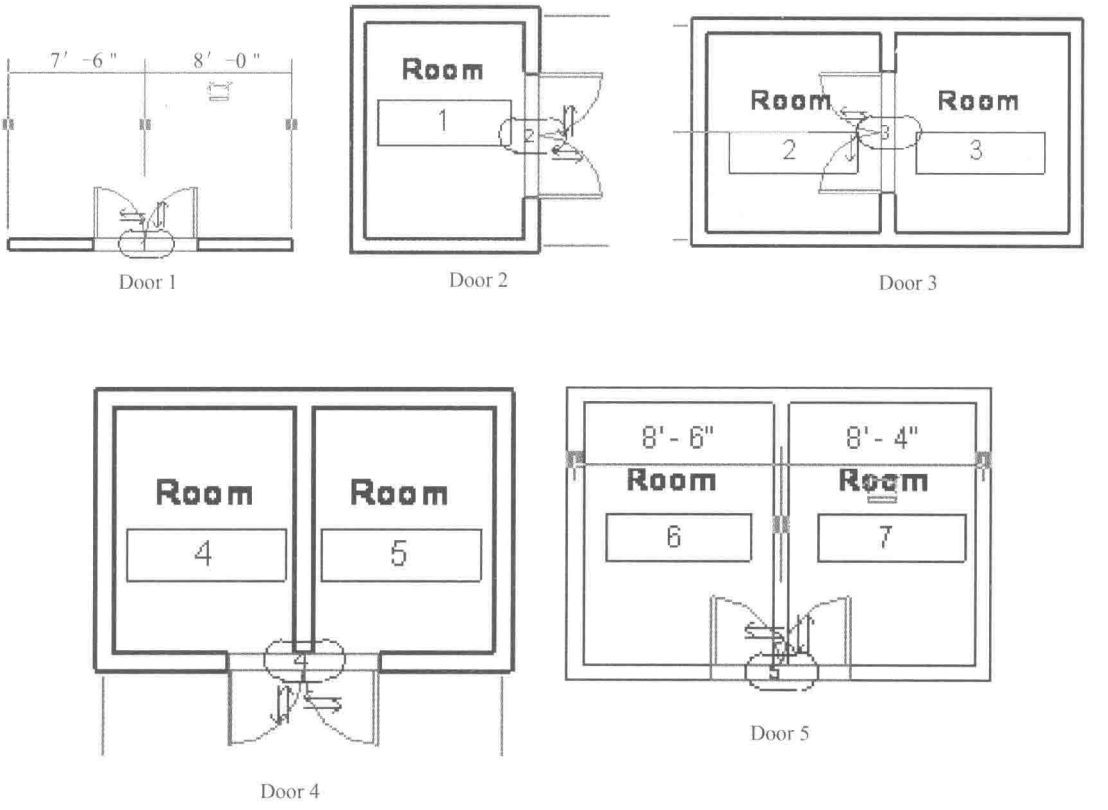


图 4-35　门和房间的关系示意图

表 4-8 给出了上图的情形中门族实例的属性 FromRoom，ToRoom 和 Room 所对应的值。

表 4-8　　门与房间的相关属性

Door	FromRoom	ToRoom	Room
Door 1	Null	null	Null
Door 2	Room 1	null	Null
Door 3	Room 3	Room 2	Room 2
Door 4	Room 4	null	Null
Door 5	null	Room 6	Room 6

4.6 线元素（CurveElement）

除了曲线 Curve，还有曲线元素 CurveElement，区别是 Curve 不是元素，而 CurveElement 是元素 Element 的子类，会被序列化到 Revit® 文件里。相关类图见图 4-36。

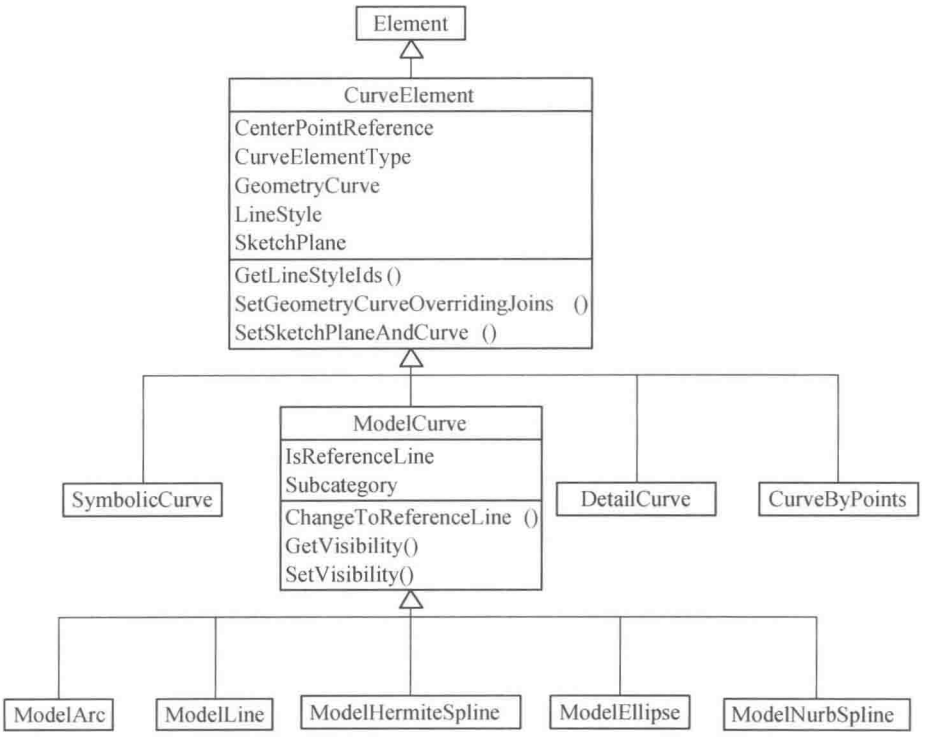


图 4-36　线元素类图

比较重要的曲线是模型线，符号线和详图线。本节主要介绍模型线。

模型线和详图线见图 4-37 和图 4-38。

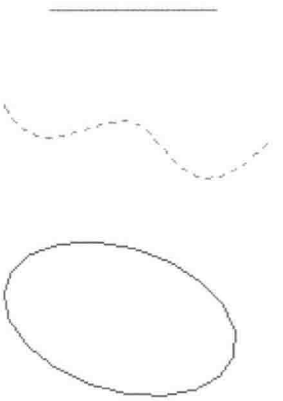


图 4-37　模型线

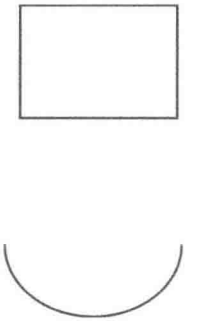


图 4-38　详图线

对应界面入口分别是：

① 工程文档里面，建筑→模型→模型线，如图 4-39 所示。

② 工程文档里面，注释→详图→详图线，如图 4-40 所示。



图 4-39　模型线

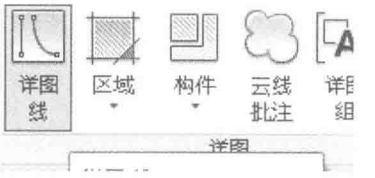


图 4-40　详图线

③ 族文档里面，注释→详图→符号线，如图 4-41 所示。

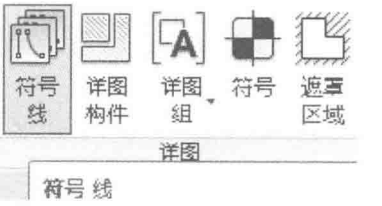


图 4-41　符号线

4.6.1　获取模型线的几何信息

可以使用 GeometryCurve 属性来获取模型线几何信息：Curve geoCurve = modelCurve.GeometryCurve。

4.6.2　获取模型线的工作平面

获取模型线的工作平面比较简单，直接使用 SketchPlane 属性：SketchPlane sketchPlane = modelCurve.SketchPlane。

4.6.3　获取和修改模型线样式

使用 LineStyle 属性可以获取和修改模型线线的样式，代码片段 4-27 展示如何修改模型线样式。

代码片段 4-27：修改线样式

```
ICollection< ElementId> styles = modelCurve.GetLineStyleIds( );
foreach (ElementId styleId in styles)
{
    if (styleId ! = modelCurve.LineStyle.Id)
```

```
    {
        using (Transaction transaction =  new Transaction(RevitDoc))
        {
            transaction.Start("Create Model Curve");
            modelCurve.LineStyle =  RevitDoc.GetElement(styleId) as GraphicsStyle;
            transaction.Commit( );
        }
        break;
    }
}
```

4.6.4 创建模型线

创建模型线可以使用 Document. Create. NewModelCurve(Curve, SketchPlane)方法。

第一个参数是 Curve，可以是任何 Curve 的子类，比如 Line，NurbSpline，Arc，Ellipse 等，那么对应创建出来的实例就是 ModelLine，ModelNurbSpline，ModelArc 和 ModelEllipse。

第二个参数是 SketchPlane 工作平面。可以使用 SketchPlane. Create 来创建工作平面。

代码片段 4-28 创建了一条直线的模型线。

代码片段 4-28:创建模型线

```
using (Transaction transaction =  new Transaction(RevitDoc))
{
    transaction.Start("Create Model Line");
    Line geoLine =  Line.CreateBound(XYZ.BasisY *  10, XYZ.BasisX *  10);
    SketchPlane modelSketch =  SketchPlane.Create(RevitDoc, RevitApp.Create.NewPlane(XYZ.
    BasisZ, XYZ.Zero));
    ModelCurve modelLine =  RevitDoc.Create.NewModelCurve(geoLine, modelSketch);
    transaction.Commit( );
}
```

代码片段 4-29 创建了一条样条曲线。

代码片段 4-29:创建样条曲线

```
using (Transaction transaction =  new Transaction(RevitDoc))
{
    SketchPlane modelSketch =  SketchPlane.Create(RevitDoc, RevitApp.Create.
    NewPlane(XYZ.BasisZ, XYZ.Zero));
    transaction.Start("Create Model NurbSpline");
    NurbSpline nurbSpline =  NurbSpline.Create(
          new List< XYZ>  { new XYZ(0, 0, 0), new XYZ(10, 0, 0), new XYZ(10, 10, 0), new XYZ
          (20, 10, 0), new XYZ(20, 20, 0) },
          new List< double>  { 0.5, 0.1, 0.3, 0.6, 0.8 });
    ModelCurve modelCurve =  RevitDoc.Create.NewModelCurve(nurbSpline, modelSketch);
    transaction.Commit( );
}
```

结果如图 4-42 所示。

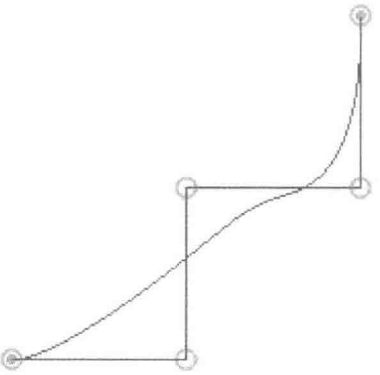


图 4-42　创建模型线结果图

4.6.5　批量创建模型线

可以使用 Document. Create. NewModelCurveArray (CurveArray, SketchPlane)方法来批量创建模型线。

第一个参数是一个线的集合，使用 Application. Create. NewCurveArray()创建一个空的集合，再使用 CurveArray. Insert()方法加入几何线，其他的方式和创建模型线一样，这里不赘述。

4.7　洞口(Opening)

4.7.1　类图

洞口类图见图 4-43。

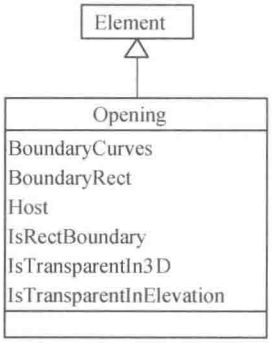


图 4-43　洞口类图

4.7.2　获取洞口的边界

如果是长方形边界，使用 BoundaryRect 属性，其他形状边界使用 BoundaryCurves 属性，用法见代码片段 4-30。

代码片段 4-30：获取洞口边界

```
if (opening.IsRectBoundary)
{
    XYZ startPoint =  opening.BoundaryRect[0];
    XYZ endPoint =  opening.BoundaryRect[1];
} else {
    foreach (Curve curve in opening.BoundaryCurves)
    {
        //遍历 Curve
    }
}
```

4.7.3　其他属性

Host 属性可以获得洞口的宿主元素。

IsTransparentIn3D 属性返回是否在三维视图透明，只能在族文档里使用。

IsTransparentInElevation 属性返回是否在里面视图透明，只能在族文档里使用。

4.7.4 创建洞口

可以使用 Document. Create. NewOpening 用来创建洞口，它有五个重载函数。

下面介绍如何在墙上创建洞口。使用重载之一 NewOpening (Wall wall, XYZ pntStart, XYZ pntEnd)函数可以在墙上创建洞口，参数意义如下：

- wall：墙对象。
- pntStart：长方形的一个顶点。
- pntEnd：长方形的另一个对角的顶点。

代码片段 4-31 展示如何在墙的正中间开一个长宽是墙的 1/3 的洞口。

代码片段 4-31：在墙上开洞口

```
Wall wall = GetElement< Wall> (185520);
LocationCurve locationCurve = wall.Location as LocationCurve;
Line location = locationCurve.Curve as Line;
XYZ startPoint = location.get_EndPoint(0);
XYZ endPoint = location.get_EndPoint(1);
Parameter wallHeightParameter = wall.get_Parameter(BuiltInParameter.WALL_USER_HEIGHT_
PARAM);
double wallHeight = wallHeightParameter.AsDouble( );
XYZ delta = (endPoint — startPoint + new XYZ(0, 0, wallHeight)) / 3;

using (Transaction transaction = new Transaction(RevitDoc))
{
    transaction.Start("Create Opening on wall");
    Opening opening = RevitDoc.Create.NewOpening(wall, startPoint + delta, startPoint +
    delta * 2);
    transaction.Commit( );
}
```

结果如图 4-44 所示。

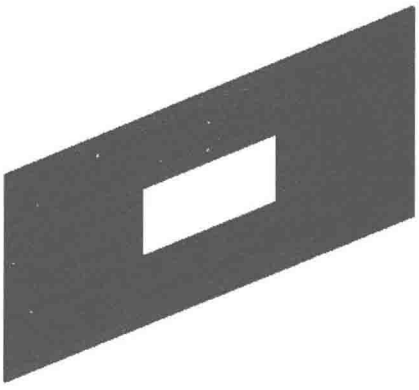


图 4-44 创建洞口结果图

注意：长方形行成的面必须和墙面平行。

第 5 章　注释（Documentation）

在 Revit® 中使用尺寸标注、文字注释、注释记号、标记和符号可以改善和增强施工图文档。通过 Revit® API 也可以相应的创建和访问这些元素。

5.1　尺寸标注(Dimension)

尺寸标注用来在项目中显示距离和尺寸的视图专有元素。所谓视图专有元素是指那些只在特定视图中显示，而不会在其他视图中显示的元素。比如尺寸标注元素，只会在它被创建的平面视图中显示。

5.1.1　尺寸标注类型

尺寸标注主要有三种类型：临时性尺寸标注(Temporary Dimensions)、永久性尺寸标注(Permanent Dimensions)和高程点标注(Spot Dimensions)。当创建或选择几何图形时，Revit® 会在构件周围显示临时尺寸标注(图 5-1)，这有利于在适当的位置放置构件。

永久性尺寸标注是特意放置的尺寸标注(图 5-2)。

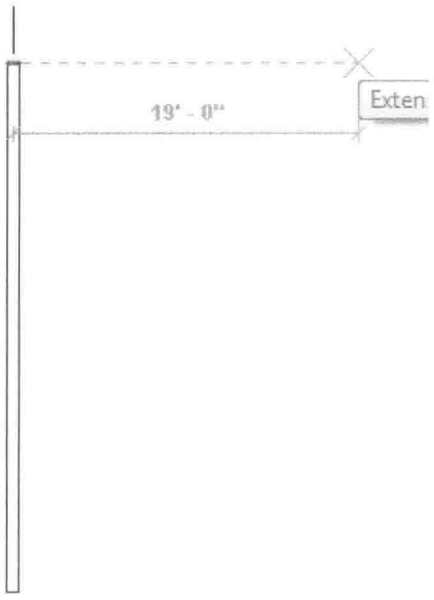


图 5-1　临时尺寸标注

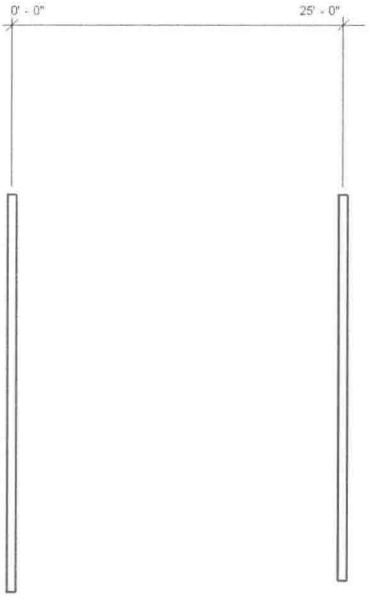


图 5-2　永久性尺寸标注

高程点标注用来显示选定点的高程(图 5-3)。

通过 Revit® API 只能访问和创建永久性尺寸标注和高程点标注。

永久性尺寸标注主要有以下几种类型：

- 对齐尺寸标注

- 线性尺寸标注
- 角度标注
- 径向尺寸标注
- 直径尺寸标注
- 弧长度尺寸标注

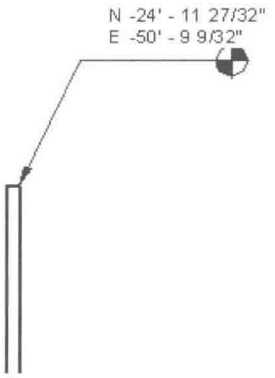


图 5-3　高程点标注

高程点标注可以采用高程点、高程点坐标或高程点坡度的形式放置。

在 Revit® API 中，由类 Autodesk. Revit. DB. Dimension 表示以上这些标注。可以通过 Dimension. DimensionShape 属性来获得标注的实际类型，它的返回值是一个枚举，有以下这些值代表相应的标准类型：

- DimensionShape. Linear(线性尺寸标注)
- DimensionShape. Angular(角度尺寸标注)
- DimensionShape. Radial(径向尺寸标注)
- DimensionShape. ArcLength(弧长度尺寸标注)
- DimensionShape. Spot(高程点标注)
- DimensionShape. Diameter(直径尺寸标注)

5.1.2　尺寸标注文字

在 Revit® 中，可以在如图 5-4 所示的对话框中自定义尺寸标注上所显示的文字(通过双击尺寸标注的文字来调出该对话框)。

可以使用实际的尺寸值再加上附加的文字字段来表示尺寸标注的文字(图 5-5)。也可以以随意的文字来替换实际尺寸值(图 5-6)。

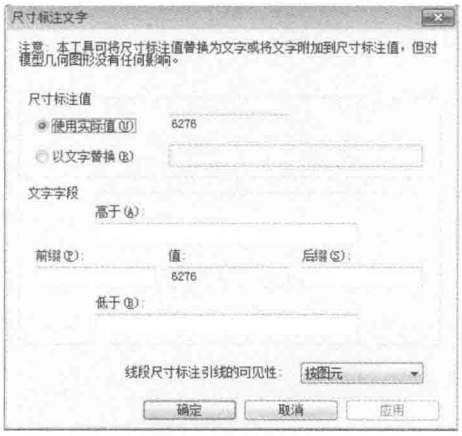


图 5-4　尺寸标注文字属性对话框

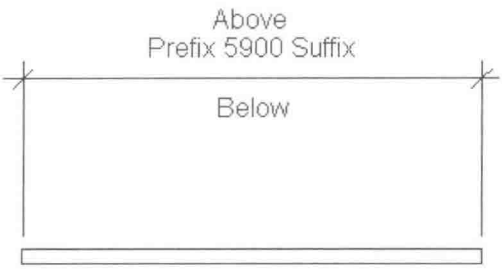


图 5-5　有附加文字的尺寸标注

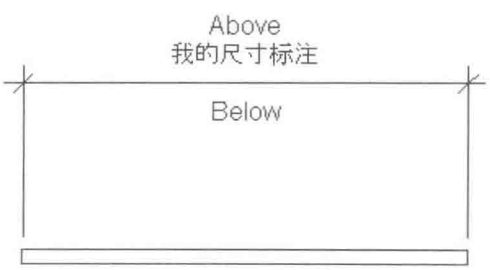


图 5-6　替换了文字的尺寸标注

Revit® API 也提供了相应的接口来读取或修改这些值(表 5-1)。

表 5-1　　尺寸标注文字的相关接口

属性	描述	访问
Dimension. Above	文字字段中的"高于"	可读写
Dimension. Below	文字字段中的"低于"	可读写
Dimension. Prefix	文字字段中的"前缀"	可读写
Dimension. Suffix	文字字段中的"后缀"	可读写
Dimension. ValueString	文字字段中的"值"	只读
Dimension. ValueOverride	替换实际尺寸值的任意文字	可读写

5.1.3　锁定尺寸标注

在 Revit® 中可以锁定或解锁永久性尺寸标注。如果处于解锁状态,则可以修改尺寸标注值,并可以在任意方向上自由移动其参照元素。如果处于锁定状态,则尺寸标注值是固定的,防止其所参照的构件间的距离发生任何变化。不能通过单击尺寸标注值来修改它。在 Revit® API 中有相应的属性 Dimension. IsLocked 来控制标注的锁定或解锁状态。

注意:当尺寸标注的类型是径向(Radial)或高程点(Spot)时,Dimension. IsLocked 属性永远返回 false。当尺寸标注是弧长度(ArcLeng)、径向(Radial)、直径(Diameter)或高程点(Spot)时,或者线性尺寸标注有多段时,该属性是无法被修改的。

5.1.4　尺寸标注的参照

可以通过 Dimension. References 来得到尺寸标注的所有参照(Autodesk. Revit. DB. Reference)。通过参照可以获得该参照所在的元素、坐标点等。

5.1.5　基线标注和同基准尺寸

基线标注和同基准尺寸都是线性尺寸标注一种类型,可以在线性尺寸的类型属性对话框中设置。

基线标注(Baseline)是指从同一基线开始测量的多个尺寸标注(图 5-7)。创建该标注时,在类型属性对话框将"标注字符串类型"设置成基线即可。

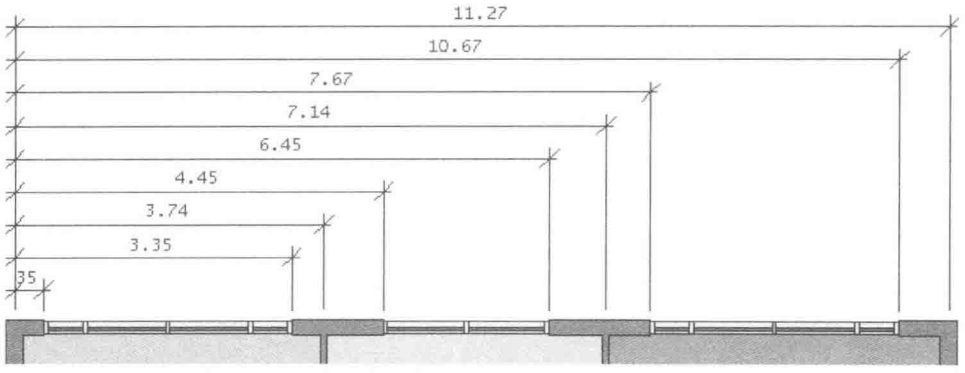


图 5-7　基线标注

同基准尺寸(Ordinate)用于测量从被称为基准的原点到某个图元的垂直距离(图 5-8)。

这些尺寸标注可以使这些对象与基准保持精确的偏移，从而防止增大误差。创建该标注时，在类型属性对话框将“标注字符串类型”设置成基线即可。

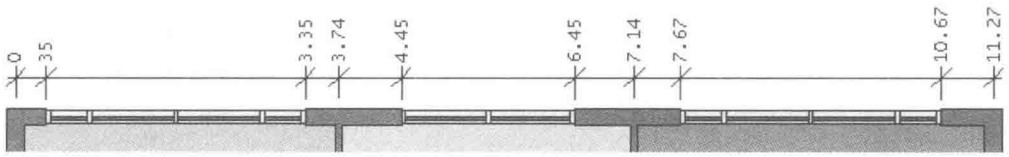


图 5-8　同基准尺寸

注意：弧长度尺寸标注是线性尺寸标注，具有可用于创建基线和同基准尺寸的参数（“标注字符串类型”和“同基准尺寸设置”），但这些参数对弧长度尺寸标注没有任何影响。

在 Revit® API 中对应这两种尺寸的类也是 Dimension。但对这种尺寸标注提供了访问各个段（Segment）的方法。Dimension. NumberOfSegments 返回该尺寸标注的段数，Dimension. Segments 返回所有段（Autodesk. Revit. DB. DimensionSegment）的集合。每个段（DimensionSegment）和参照（Reference）有一一对应的关系，第一个段在第一个和第二个参照之间，第 N 个段在第 N 个和第 $N+1$ 个参照之间。对每个 DimensionSegment 提供了与 Dimension 类似的属性访问其标注的值（表 5-2）。

表 5-2　　尺寸段（DimensionSegment）的接口

属性	描述	访问
DimensionSegment. Above	文字字段中的“高于”	可读写
DimensionSegment. Below	文字字段中的“低于”	可读写
DimensionSegment. Prefix	文字字段中的“前缀”	可读写
DimensionSegment. Suffix	文字字段中的“后缀”	可读写
DimensionSegment. ValueString	文字字段中的“值”	只读
DimensionSegment. ValueOverride	替换实际尺寸值的任意文字	可读写
DimensionSegment. IsLocked	锁定该尺寸标注段	可读写
DimensionSegment. Origin	尺寸标注段的原点	只读

5.1.6　尺寸标注的创建

Revit® API 提供了以下方法来创建文档级别的线性尺寸标注和对齐尺寸标注。

1）创建线性尺寸标注

- Autodesk. Revit. Creation. Document. NewDimension（View view, Line line, ReferenceArray references）

其中，参数 view 是尺寸标注所要创建在的视图，line 表示线性尺寸的直线，references 是绑定的几何参照。

- Autodesk. Revit. Creation. Document. NewDimension（View view, Line line, ReferenceArray references, DimensionType dimensionType）

其中，参数 view 是尺寸标注所要创建在的视图，line 表示线性尺寸的直线，references 是绑定的几何参照，dimensionType 是尺寸标注的族类型。

2) 创建对齐尺寸标注

- Autodesk. Revit. Creation. Document. NewAlignment (View view, Reference reference1, Referencereference2)

其中,参数 view 是尺寸标注所要创建在的视图,reference1 和 reference2 是绑定的几何参照。

另外,以下方法可以创建族文档级别的各种尺寸标注。

3) 创建角度尺寸标注

- Autodesk. Revit. Creation. FamilyItemFactory. NewAngularDimension (View view, Arc arc, Reference firstRef, Reference secondRef)

其中,参数 view 是尺寸标注所要创建在的视图,arc 表示所要标注的圆弧,firstRef 和 secondRef 是尺寸标注的两个参照,它们必须和圆弧垂直。

- Autodesk. Revit. Creation. FamilyItemFactory. NewAngularDimension (View view, Arc arc, Reference firstRef, Reference secondRef, DimensionType dimensionType)

其中,参数 view 是尺寸标注所要创建在的视图,arc 表示所要标注的圆弧,firstRef 和 secondRef 是尺寸标注的两个参照,它们必须和圆弧垂直。dimensionType 是尺寸标注的族类型。

4) 创建弧长度尺寸标注

- Autodesk. Revit. Creation. FamilyItemFactory. NewArcLengthDimension (View view, Arc arc, Reference arcRef, Reference firstRef, Reference secondRef)

其中,参数 view 是尺寸标注所要创建在的视图,arc 表示所要标注的圆弧,arcRef 表示所要标注的圆弧的几何参照,firstRef 和 secondRef 是尺寸标注的两个参照,它们必须和圆弧相交。

- Autodesk. Revit. Creation. FamilyItemFactory. NewArcLengthDimension (View view, Arc arc, Reference arcRef, Reference firstRef, Reference secondRef, DimensionType dimensionType)

其中,参数 view 是尺寸标注所要创建在的视图,arc 表示所要标注的圆弧,arcRef 表示所要标注的圆弧的几何参照,firstRef 和 secondRef 是尺寸标注的两个参照,它们必须和圆弧相交。dimensionType 是尺寸标注的族类型。

5) 创建直径尺寸标注

- Autodesk. Revit. Creation. FamilyItemFactory. NewDiameterDimension (View view, Reference arcRef, XYZ origin)

其中,参数 view 是尺寸标注所要创建在的视图,arcRef 表示所要标注的圆弧的几何参照,origin 和圆心连线表示尺寸标注所在的线。

6) 创建线性尺寸标注

- Autodesk. Revit. Creation. FamilyItemFactory. NewLinearDimension(View view, Line line, ReferenceArray references)

其中,参数 view 是尺寸标注所要创建在的视图,line 表示线性尺寸的直线,references 是绑定的几何参照。

- Autodesk. Revit. Creation. FamilyItemFactory. NewLinearDimension(View view, Line line, ReferenceArray references, DimensionType dimensionType)

其中,参数 view 是尺寸标注所要创建在的视图,line 表示线性尺寸的直线,references 是绑定的几何参照,dimensionType 是尺寸标注的族类型。

7) 创建径向尺寸标注

- Autodesk. Revit. Creation. FamilyItemFactory. NewRadialDimension(View view, Reference arcRef, XYZ origin)

其中,参数 view 是尺寸标注所要创建在的视图,arcRef 表示所要标注的圆弧的几何参照,origin 和圆心连线表示尺寸标注所在的线。

- Autodesk. Revit. Creation. FamilyItemFactory. NewRadialDimension(View view, Reference arcRef, XYZ origin, DimensionType dimensionType)

其中,参数 view 是尺寸标注所要创建在的视图,arcRef 表示所要标注的圆弧的几何参照,origin 和圆心连线表示尺寸标注所在的线,dimensionType 是尺寸标注的族类型。

代码片段 5-1 展示了如何创建线性尺寸标注。

代码片段 5-1:创建线性尺寸标注

```
Transaction transaction = new Transaction(m_revit.Application.ActiveUIDocument.Document,
    "添加标注");
transaction.Start( );
//取得墙,给墙的两个端点创建线性尺寸
for (int i = 0; i < m_walls.Count; i++ )
{
    Wall wallTemp = m_walls[i] as Wall;
    if (null = = wallTemp)
    {
    continue;
}

//取得位置线
Location location = wallTemp.Location;
LocationCurve locationline = location as LocationCurve;
if (null = = locationline)
{
    continue;
}

Line newLine = null;

//取得参考
ReferenceArray referenceArray = new ReferenceArray( );

AnalyticalModel analyticalModel = wallTemp.GetAnalyticalModel( );
IList < Curve > activeCurveList = analyticalModel. GetCurves (AnalyticalCurveType.
ActiveCurves);
foreach (Curve aCurve in activeCurveList)
{
```

```
    //从分析模型中找到不垂直的线
    if (aCurve.GetEndPoint(0).Z = = aCurve.GetEndPoint(1).Z)
        newLine = aCurve as Line;
    if (aCurve.GetEndPoint(0).Z ! = aCurve.GetEndPoint(1).Z)
    {
        AnalyticalModelSelector amSelector = new AnalyticalModelSelector(aCurve);
        amSelector.CurveSelector = AnalyticalCurveSelector.StartPoint;
        referenceArray.Append(analyticalModel.GetReference(amSelector));
    }
    if (2 = = referenceArray.Size)
        break;

if (referenceArray.Size ! = 2)
{
    m_errorMessage + = "Did not find two references";
    return false;
}
try
{
    //创建尺寸
    Autodesk.Revit.UI.UIApplication app = m_revit.Application;
    Document doc = app.ActiveUIDocument.Document;

    Autodesk.Revit.DB.XYZ p1 = new XYZ(
        newLine.GetEndPoint(0).X + 5,
        newLine.GetEndPoint(0).Y + 5,
        newLine.GetEndPoint(0).Z);
    Autodesk.Revit.DB.XYZ p2 = new XYZ(
        newLine.GetEndPoint(1).X + 5,
        newLine.GetEndPoint(1).Y + 5,
        newLine.GetEndPoint(1).Z);

    Line newLine2 = Line.CreateBound(p1, p2);
    Dimension newDimension = doc.Create.NewDimension(
        doc.ActiveView, newLine2, referenceArray);
}
catch (Exception ex)
{
    m_errorMessage + = ex.ToString( );
    return false;
}
}
transaction.Commit( );
```

5.1.7　高程点标注

高程点标注（图 5-9）是尺寸标注的子类型（Autodesk. Revit. DB. SpotDimension）。除了基类 Dimension 提供的这些访问属性外，它还有一个额外的 SpotDimension. Location 属性来访问它所在的位置。

Revit® API 提供了以下方法来创建高程点标注和高程点坐标标注：

- Autodesk. Revit. Creation. Document. NewSpotElevation (View view, Reference reference, XYZ origin, XYZ bend, XYZ end, XYZ refPt, bool hasLeader)

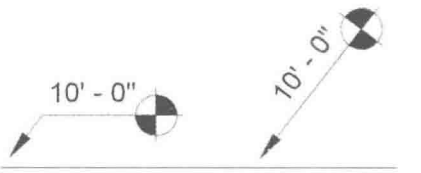


图 5-9 高程点标注

其中,参数 view 是高程点标注所要创建在的视图,reference 是其几何参照,origin 是标注的起点,bend 是弯曲点,end 是标注的终点,refPt 是标注所要测量的点,hasLeader 表示是否有箭头。

- Autodesk. Revit. Creation. Document. NewSpotCoordinate(View view, Reference reference, XYZ origin, XYZ bend, XYZ end, XYZ refPt, bool hasLeader)

其中,参数 view 是高程点标注所要创建在的视图,reference 是其几何参照,origin 是标注的起点,bend 是弯曲点,end 是标注的终点,refPt 是标注所要测量的点,hasLeader 表示是否有箭头。



图 5-10 水平和倾斜的文字注释

5.2 文字注释(Text)

可以在视图中加入文字注释,这些注释在纸空间中测量而且自动随视图一起缩放。图 5-10 显示了两个文字注释,一个水平放置,一个倾斜放置。

5.2.1 文字注释的属性

在 Revit® API 中,类 TextNote 代表文字注释元素。现有的 API 可以访问表 5-3 所列接口。

表 5-3 文字注释(TextNote)的相关接口

接口	功能	访问
TextNote. Align	文字对齐方式,可以是以下几种: • TextAlignFlags. TEF_ALIGN_CENTER(居中) • TextAlignFlags. TEF_ALIGN_LEFT(左对齐) • TextAlignFlags. TEF_ALIGN_RIGHT(右对齐)	只读
TextNote. Height	文字注释的高度	只读
TextNote. Width	文字注释的宽度	可读写
TextNote. LineWidth	文字注释的边框和引线的宽度	只读
TextNote. Text	文字注释中的文字	可读写
TextNote. BaseDirection	文字注释的水平方向	只读
TextNote. UpDirection	文字注释的垂直方向,它和 BaseDirection 一起决定了文字的显示方向是水平的,还是倾斜的,还是垂直的	只读
TextNote. Symbol	文字注释的族类型	只读
TextNote. Coord	文字注释的坐标	可读写
TextNote. Leaders	文字注释的箭头集合,可以通过 Leader 对象来访问其属性,如弯头点(Leader. Elbow)、端点(Leader. End)等	只读
TextNote. TextNoteType	文字注释的类型	可读写

另外，还可以通过参数(Parameter)来访问和修改所需要的内容。

5.2.2　文字注释的创建

通过以下方法可以创建文字注释。

- Autodesk. Revit. Creation. Document. NewTextNote (View pView, XYZ origin, XYZ baseVec, XYZ upVec, double lineWidth, TextAlignFlags textAlign, string strText)

其中，pView 是文字注释所要创建在的视图，origin 是其原点，baseVec 和 upVec 决定了其水平和垂直方向，lineWidth 是线宽，textAlign 是文字的对齐方式，strText 是文字内容。

- Autodesk. Revit. Creation. Document. NewTextNote(View pView, XYZ origin, XYZ baseVec, XYZ upVec, double lineWidth, TextAlignFlags textAlign, TextNoteLeaderTypes leaderType, TextNoteLeaderStyles leaderStyle, XYZ leaderEnd, XYZ leaderElbow, string strText)

其中，pView 是文字注释所要创建在的视图，origin 是其原点，baseVec 和 upVec 决定了其水平和垂直方向，lineWidth 是线宽，textAlign 是文字的对齐方式，leaderType 是箭头的类型(直线型或弧形)，leaderStyle 是箭头的样式(一段直线型、一段弧线型或两段直线型)，leaderEnd 是箭头的端点，leaderElbow 是箭头的弯曲点，strText 是文字内容。

5.3　详图(Detail)

Revit® 可以将项目构造为现实世界中物理对象的数字表示形式。但是，不是每一个构件都需要进行三维建模。建筑师和工程师可创建标准详图，以说明如何构造较大项目中的材质。详图搭建建筑设计和实际建筑之间的桥梁，并将有关如何实现设计的信息传递给施工人员和承包人。详图是对项目的重要补充，因为它们显示了材质应该如何相互连接。

有关详图视图的创建和访问可以参考第 8 章“视图(View)”。本节将介绍在详图中使用的以及可以通过 API 访问的详图线和区域。

5.3.1　详图线

详图线与其他注释一样，也是基于视图的。详图线是在视图的草图平面中绘制的。

详图线相对比较简单，类 Autodesk. Revit. DB. DetailLine 代表详图线，通过 Revit® API 可以访问表 5-4 所列的接口。

表 5-4　　详图线(DetailLine)的相关接口

接口	功能	访问
DetailLine. GeometryCurve	详图线相对应的几何线	可读写
DetailLine. LineStyle	线样式	可读写

详图线的创建也比较简单，在普通文档和族文档中都可以创建：

- Autodesk. Revit. Creation. ItemFactoryBase. NewDetailCurve(View view, Curve geometryCurve)

其中，参数 view 是所要创建的详图线所在的视图，geometryCurve 是其对于的几何线。

5.3.2　区域

Revit® 中，区域有填充区域(Filled Region)和遮罩区域(Masking Region)两种。

填充区域是由某个样式的边界线封闭起来的、内部用某个样式填充的区域。它是基于

二维视图的，可用于在详图视图中定义填充区域或将填充区域添加到注释族中。

遮罩区域提供了一种在视图中隐藏图元的方法，可以创建二维遮罩区域和三维遮罩区域。可以在创建二维族(注释或详图)时，在项目中和族编辑器中创建二维遮罩区域。可以在创建模型族时，在族编辑器中创建三维遮罩区域。

在 Revit® API 中，填充区域和遮罩区域都由同一个类 Autodesk. Revit. DB. FilledRegion 表示。属性 FilledRegion. IsMasking 如果为 true 则表示该区域是遮罩区域，false 则表示是填充区域。

Revit® API 还提供了函数来访问和修改区域的线样式。函数 FilledRegion. GetValidLineStyleIdsForFilledRegion(Document document)可以获得文档中所有可用的区域线样式。函数 FilledRegion. SetLineStyleId(ElementId lineStyleId)则可以用来修改区域的线样式。函数 FilledRegion. GetBoundaries()则可以获得组成区域包围盒的各个线段。另外，还可以访问区域的参数来获得面积等重要信息。

区域的创建则可以用静态函数 FilledRegion. Create(Document document, ElementId typeId, ElementId viewId, IList＜CurveLoop＞ boundaries)。其中，参数 document 是文档对象，typeId 是区域的类型对象的 Id，可以通过 FilledRegion. IsValidFilledRegionTypeId (Document document, ElementId typeId)函数组合的过滤器来得到(参见“3. 3 元素过滤器”)，boundaries 则是所有创建区域的包围盒的线圈。

5. 4 标记(Tag)

标记是用于在图纸中识别元素的注释(图 5-11)。与标记相关联的属性会显示在明细表中。

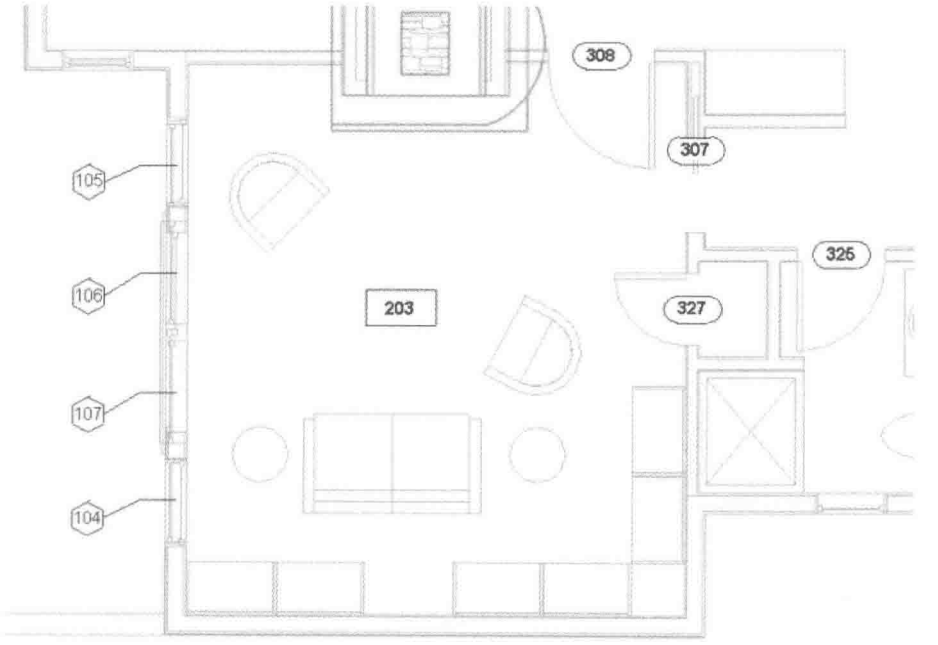


图 5-11 门、窗和房间的标记

族库中的每个类别都有一个标记。一些标记会随默认的 Revit®样板自动载入，而另一

些则需要手动载入。如果需要，可以在族编辑器中创建自己的标记。

5.4.1　类别标记和材质标记

在 Revit® 中，类别标记有按类别标记和多类别标记两种。按类别标记是按元素的类别进行标记，而多类别标记则是按共享参数进行标记，可以在不同类别上使用相同的标记，需要载入相应的族。

材质标记则是按照元素的材质进行标记。

在 Revit® API 中，以上这些标记都由 Autodesk. Revit. DB. IndependentTag 表示。属性 IndependentTag. IsMulticategoryTag 和属性 IndependentTag. IsMaterialTag 分别表示该标记是多类别标记还是材质标记，如果都为 false，则为按类别标记。

表 5-5 列出了 IndependentTag 其他重要的属性。

表 5-5　　IndependentTag 的其他属性

属性	功能	访问
IndependentTag. HasLeader	是否有箭头	可读写
IndependentTag. LeaderElbow	箭头拐点位置	可读写
IndependentTag. LeaderEnd	箭头终点位置	可读写
IndependentTag. LeaderEndCondition	箭头终点属性，紧贴标记物（LeaderEndCondition. Attatched）或自由放置（LeaderEndCondition. Free）	可读写
IndependentTag. TaggedElementId	所标记的元素 LinkElementId，是连接文件里的元素 Id，如果该元素被删除了，那么它的值都是 InvalidElementId	只读
IndependentTag. IsOrphaned	True 表示该标记所指的元素已无效，比如被删除了	只读
IndependentTag. TaggedLocalElementId	所标记的本地元素的 Id	只读
IndependentTag. TagHeadPosition	标记的位置	可读写
IndependentTag. TagOrientation	标记的朝向，可以是水平（TagOrientation. Horizontal）或者是垂直（TagOrientation. Vertical）	可读写
IndependentTag. TagText	标记的文字内容	只读

IndependentTag 的创建可以用函数 Autodesk. Revit. Creation. Document. NewTag (View dbview, Element elemToTag, bool addLeader, TagMode tagMode, TagOrientation tagOrientation, XYZ pnt)。

其中，参数 dbview 是所要创建在的视图，elemToTag 是要放标记的元素，addLeader 是否要加箭头，tagMode 是标记的类型，分别是类别标记（TagMode. TM_ADDBY_CATEGORY）、多类别标记（TagMode. TM_ADDBY_MULTICATEGORY）或者材质标记（TagMode. TM_ADDBY_MATERIAL），tagOrientation 是标记的方向，分别是垂直（TagOrientation. Horizontal）或者水平（TagOrientation. Vertical）。

5.4.2　注释记号（Keynote）

注释记号和类别记号、材质记号一样，在 Revit® API 中由 IndependentTag 表示。各个属性和参数的访问也一样，但没有提供创建函数。

5.4.3　房间、空间和面积标记

房间、空间和面积标记是可在平面视图和剖面视图中添加和显示的注释元素。房间标

记是标记房间的，可以显示相关参数的值，如房间编号、房间名称、计算的面积和体积等参数。空间标记则是用于空间的，空间专用于 MEP 规程，以进行体积分析。它们包含了用于维护其所在区域的相关信息的参数。该信息用于执行热负荷和冷负荷分析。面积标记显示了面积边界内的总面积。

在 Revit® API 中，这三个类型分别由类 Autodesk. Revit. DB. RoomTag、Autodesk. Revit. DB. SpaceTag 和 Autodesk. Revit. DB. AreaTag 表示。它们都继承自 Autodesk. Revit. DB. SpatialElementTag，共同的一些主要属性见表 5-6。

表 5-6　　SpatialElementTag 的属性表

属性	功能	访问
SpatialElementTag. HasLeader	是否有箭头	可读写
SpatialElementTag. Location	标记的位置	只读
SpatialElementTag. View	标记所在视图	只读

它们各自还有一些相应的属性，见表 5-7、表 5-8 和表 5-9。

表 5-7　　RoomTag 的属性表

属性	功能	访问
RoomTag. Room	房间标记所在的房间	只读
RoomTag. RoomTagType	房间标记的族类型	可读写

表 5-8　　SpaceTag 的属性表

属性	功能	访问
SpaceTag. Space	空间标记所标记的空间	只读
SpaceTag. SpaceTagType	空间标记的族类型	可读写

表 5-9　　AreaTag 的属性表

属性	功能	访问
AreaTag. Area	面积标记所标记的面积	只读
AreaTag. AreaTagType	面积标记的族类型	可读写

5.4.4　符号(Symbol)

符号是注释元素或其他对象的图形表示。有时也将符号称为标记。

如图 5-12 所示的符号图例标识了在一系列施工图设计文件中使用的注释符号。

在 Revit® API 中，符号由类 Autodesk. Revit. DB. AnnotationSymbol 表示，提供了如表 5-10 所示的重要函数。



图 5-12　各种注释符号图

表 5-10　　AnnotationSymbol 的方法及功能表

方法	功能
AnnotationSymbol. addLeader	添加箭头
AnnotationSymbol. removeLeader	删除箭头
AnnotationSymbol. duplicate	复制符号

第 6 章　几何（Geometry）

6.1　概述

在 Autodesk. Revit. DB 命名空间里包含了一些几何图形相关的类型，它们在 API 中用于几何图形的表示和处理。从基类继承的情况分，API 提供了三大种几何类型来描述和存储几何信息：

- 几何基元类：包括所有从 GeometryObject 派生出来的子类。
- 几何辅助类：包括一些从 APIObject 派生出来的几何相关的子类和一些值类型。
- 几何集合类：包括一些实现了 IEnumerable 或者 IEnumerator 接口的几何相关的类型。

本章将介绍如何使用各种图形相关的类型，如何从元素中获取几何数据，如何让一个元素进行图形变换等。

6.2　实战：获取一堵墙的几何数据

本例将演示如何从一堵墙上获取几何数据：

- 获取墙的边
- 获取墙的面

注意：作为从元素中获取几何数据的例子，这个例子有一定的限制，几何实例 Instance 没有包括在里面，像墙里面的放样 Sweep 就没有涉及到。这个例子的目的是让读者对如何获取几何数据有一个简单的印象，并没有包括所有的条件。如果想进一步了解如何从元素中获取几何数据，请参考“6. 6 实战：获取一根梁的几何数据”。

6.2.1　创建几何选项

为了获取墙的几何信息，必须先创建一个几何选项，提供一些详细的自定义参数选项。几何选项将决定从元素的几何属性 Element. Geometry 里面取到的几何信息。下面的代码片段 6-1 创建了几何选项 option，并设置了其选项计算几何引用为打开，视图的详细程度设为 Fine。

代码片段 6-1：几何

```
public OptionsGetGeometryOption( )
    {
Autodesk.Revit.DB.Options option=  this.Application.Create.NewGeometryOptions( );
    option.ComputeReferences =  true;  //打开计算几何引用
    option.DetailLevel =  ViewDetailLevel.Fine;  //视图详细程度为最好
return option;
}
```

注意：想了解更多的信息，请参考“6.4.3 几何选项（Options）”。

6.2.2　获取面和边

墙的几何数据主要是一个由面和边组成的几何实体，由下面的步骤获取：

① 通过墙类型的几何属性 Geometry 获取 GeometryElement 的实例。这个实例里面存储了所有相关的几何对象，如实体、线等。

② 遍历 GeometryElement 实例来得到一个几何实体 solid，这个几何实体 solid 的 Faces 和 Edges 属性里面包含了所有的几何面和边。

③ 遍历 Faces 属性得到所有几何面。

④ 遍历 Edges 属性得到所有的几何边。

例子的代码片段 6-2 显示如下。

代码片段 6-2：几何

```
public void GetWallGeometry( )
{
Document doc =  this.ActiveUIDocument.Document;
Wall aWall =  doc.GetElement(new ElementId(186388)) as Wall;

Options option =  GetGeometryOption( );//创建几何选项
    Autodesk.Revit.DB.GeometryElement geomElement =  aWall.get_Geometry(option);
    foreach (GeometryObject geomObj in geomElement)
    {
        Solid geomSolid =  geomObj as Solid;
        if (null ! =  geomSolid)
        {
            foreach (Face geomFace in geomSolid.Faces)
            {
                // 得到墙的面
            }
            foreach (Edge geomEdge in geomSolid.Edges)
            {
                // 得到墙的边
            }
        }
    }
}
```

6.3　几何基元类

几何基元类在 API 中描述图形表示，由基类 GeometryObject 派生，主要有如下的类型：

① 轮廓（Profile）：轮廓是可用来生成形状的单条线、一串连接起来的线或闭合的环。可以通过操纵轮廓来修改形状的几何图形。

② 面（Face）：三维空间的实体面。其子类型有：平面（PlanarFace），规则曲面（RuledFace），旋转面（RevolvedFace），圆锥面（ConicalFace），圆柱面（CylindricalFace），Hermite 曲面（HermiteFace）。

③ 边(Edge):三维空间实体的边。

④ 线(Curve):参数曲线。其子类有:直线(Line),圆弧线(Arc),椭圆线(Ellipse),螺旋线(CylindricalHelix),Hermite 样条曲线(HermiteSpline),NURBS 样条曲线(NurbSpline)。

⑤ 点(Point):三维空间的点。

⑥ 几何元素(GeometryElement):一个元素的几何表示,包含了所有的几何信息。

⑦ 几何实例(GeometryInstance):一个类型图元的实例,可以取得与该实例相关的类型图元的几何信息。

⑧ 网格(Mesh):三角化网格用于描述三维面的形状。

⑨ 实体(Solid):三维实体。

6.3.1 几何实例(GeometryInstance)

几何实例表示了储存在 Revit® 中默认配置的几何组合,通过各种变换到适当的位置成为一个元素的属性。最普遍的情况是在族实例中应用几何实例。Revit® 使用几何实例来存储一份特定族元素类型的几何拷贝,然后在多个族实例中重复使用它。但是并不是所有的族实例都会有几何实例。当由于局部相连,相交,以及其他种种因素影响到实例位置时,Revit® 只需要唯一一份特定族元素类型的几何表示,这种情况就不会有几何实例,而是用实体来进行几何表示。

几何实例中有 SymbolGeometry 属性,该属性是生成这个几何实例的类型的几何表示,它使用的是局部坐标系。几何实例也提供了一个 Transform 属性,表示了从类型的局部坐标系到实例的世界坐标系的坐标变换。

几何实例也提供了 GetInstanceGeometry()和 GetSymbolGeometry()方法来分别获取其族实例的几何元素和族类型的几何元素。

一般来说,从几何实例中可以得到三种类型的数据:

- 线(Curve):基类线或者子类的线。
- 实体(Solid):包含了面和边。
- 几何实例(Instance):另一个几何实例,用来构造本几何实例。

用户有时候需要使用 Transform 属性对取到的几何数据进行坐标变换。更详细的信息,请参考"6.4.1 几何变换(Transform)"。

下面有两个示例来演示如何从一个几何实例得到它的几何数据。

1) 示例 1

首先从族实例中获取到它的几何元素,然后从几何元素中找到其几何实例,再从几何实例的 SymbolGeometry 属性中获取族元素类型的几何元素,从类型的几何元素中取到所有的线,然后进行坐标变换。参见代码片段 6-3。

代码片段 6-3:几何

```
public void GetInstanceGeometry_Curve( )
{
    Document doc = this.ActiveUIDocument.Document;
      FamilyInstance familyInstance = doc.GetElement(new ElementId(187032)) as
      FamilyInstance;
```

```
    Options option = GetGeometryOption( );
    Autodesk.Revit.DB.GeometryElement geomElement = familyInstance.get_Geometry(option);

    foreach (GeometryObject geomObj in geomElement)
    {
        GeometryInstance geomInstance = geomObj as GeometryInstance;
        if (null ! = geomInstance)
        {
            foreach (GeometryObject instObj in geomInstance.SymbolGeometry)
            {
                Curve curve = instObj as Curve;
                if (null ! = curve)
                {
                    // 把取到的线变换到实例的坐标系中
                    curve = curve.CreateTransformed(geomInstance.Transform);
                    // …
                }
            }
        }
    }
}
```

注意：从一个族实例中找线是不常见的行为，如果需要从实例中拿到一根线，请编辑该族，然后添加一根线。

2）示例 2

首先获取族元素类型的几何元素，步骤和示例 1 的一样。然后从类型的几何元素中取到它的实体，遍历实体中的所有面和边，把面转化为网格后对网格的顶点进行坐标变换；同样，对获取到的边进行镶嵌细分后的点进行坐标变换。见代码片段 6-4。

代码片段 6-4：几何

```
public void GetInstanceGeometry_Solid( )
{
Document doc = this.ActiveUIDocument.Document;
FamilyInstance familyInstance = doc.GetElement(new ElementId(187758)) as FamilyInstance;
Options option = GetGeometryOption( );
Autodesk.Revit.DB.GeometryElement geomElement = familyInstance.get_Geometry(option);

foreach (GeometryObject geomObj in geomElement)
{
GeometryInstance geomInstance = geomObj as GeometryInstance;
if (null ! = geomInstance)
{
foreach (GeometryObject instObj in geomInstance.SymbolGeometry)
{
```

```
Solid solid = instObj as Solid;
if (null == solid || 0 == solid.Faces.Size || 0 == solid.Edges.Size)
{
continue;
}
Transform instTransform = geomInstance.Transform;
//从实体 Solid 获取面和边,然后对点进行变换
foreach (Face face in solid.Faces)
{
Mesh mesh = face.Triangulate( );
foreach (XYZ ii in mesh.Vertices)
{
XYZ point = ii;
XYZ transformedPoint = instTransform.OfPoint(point);
}
}
foreach (Edge edge in solid.Edges)
{
foreach (XYZ ii in edge.Tessellate( ))
{
XYZ point = ii;
XYZ transformedPoint = instTransform.OfPoint(point);
}
}
}
}
}
}
```

注意:当获取门,窗和其他族元素的几何数据时,步骤是相似的。更多获取几何数据的信息,请参考"6.6 实战:从一根梁中获取梁的几何数据"。

6.3.2 网格(Mesh)

Revit®的网格是多边形网格,由一系列顶点以及对应的一系列三角形组成,用于对三维面的表示。代码片段 6-5 演示了 API 如何由很多小三角形来表示一个体量(放样)的几何空间面。首先获取放样 Sweep 的几何元素,从其中获取实体和构成实体的面。对面进行三角面片化后形成对应的网格,遍历网格的所有三角形并收集所有的顶点,最后把这些顶点在文档中描绘出来。

代码片段 6-5:几何

```
public void DrawMesh( )
{
Document doc = this.ActiveUIDocument.Document;
Transaction transaction = new Transaction(doc, "Draw Mesh");
transaction.Start( );

Sweep sweep = doc.GetElement(new ElementId(2311)) as Sweep;
```

```
Options option = GetGeometryOption( );
Autodesk.Revit.DB.GeometryElement geomElement = sweep.get_Geometry(option);

foreach (GeometryObject geomObj in geomElement)
{
Solid geomSolid = geomObj as Solid;
if (null ! = geomSolid)
{
foreach (Face geomFace in geomSolid.Faces)
{
//对面进行三角面片化形成一个网格
Mesh mesh = geomFace.Triangulate( );
for (int i = 0; i < mesh.NumTriangles; i+ + )
{
MeshTriangle triangular = mesh.get_Triangle(i);
//定义 XYZ 列表来存放三角形的顶点
List< XYZ> triangularPoints = new List< XYZ> ( );
for (int n = 0; n < 3; n+ + )
{
XYZ point = triangular.get_Vertex(n);
triangularPoints.Add(point);
}
//调用方法把所有三角形在文档中描绘出来
DrawTriangle(doc, triangularPoints);
}
}
}
}

transaction.Commit( );
}
```

图 6-1 左边的物体表示一个 Revit® 中的体量，右边的物体表示了上面示例画出来的由三角形组成的网格。

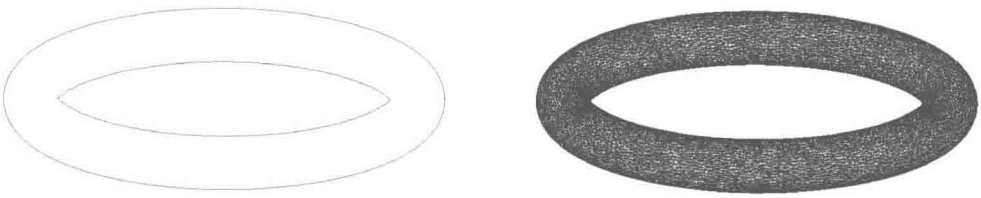


图 6-1　网格

注意：为了满足显示的需要，图中所用得公差近似值是在 Revit® 内部中定义的。

6.3.3　实体(Solid)

实体类型定义了一个包含了面和边的三维几何实体，如立方体和长方体，同时可以从它的属性中获取对应的表面积和体积。实体的属性见表 6-1。

表 6-1　　实体的属性和描述表

属性	描述
Edges	实体的边
Faces	实体的面
SurfaceArea	实体的表面积
Volume	实体的体积

有时 API 可以取到没有任何边和面的空实体，使用前请先检查对应的属性来确保获取想要的边和面。参见代码片段 6-6。

代码片段 6-6：几何

```
// GeometryObject geoObj = GetGeometryObject( );
Solid solid = geoObj as Solid;
if(null ! = solid && 0 ! = solid.Faces.Size)
{
//先判断再使用
}
```

6.4　几何辅助类

API 有一些辅助类来帮助表示某些元素的几何信息，比如视图的裁剪盒是由 BoundingBoxXYZ 对象来定义的。

① BoundingBoxXYZ：三维的长方体。定义了指定坐标系统的变换(Transform)，最小坐标值(Min)，以及最大坐标值(Max)。

② Transform：仿射空间的一个变换。

③ Reference：Revit®模型中的一个几何对象的稳定引用，一般用于创建元素，比如尺寸标注等。

④ Plane：几何平面对象。用户可以构建一个平面对象，设置对应的原点和向量。

⑤ Options：解析几何的用户参数选择。这个类决定了几何属性的数据。

⑥ XYZ：三维空间的坐标。通常表示一点或者一个向量。

⑦ UV：二维空间的坐标。通常表示一个表面的参量，在实际应用中，可以理解为二维空间的一点或者一个向量。

⑧ BoundingBoxUV：二维的长方形，平行于坐标轴。

图 6-2 表示了这些辅助类之间的关系。

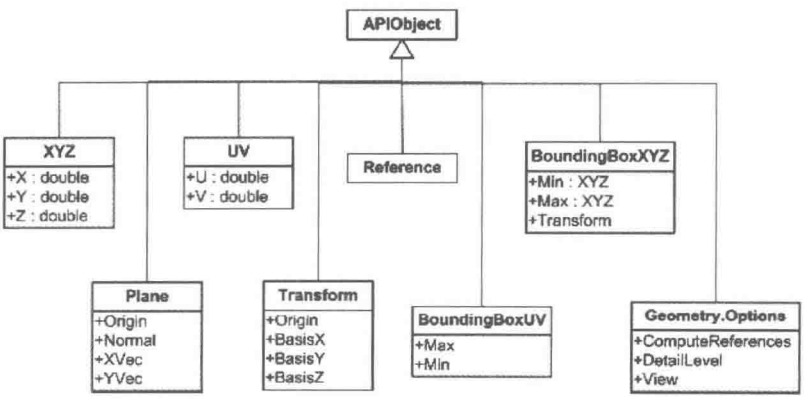


图 6-2　几何辅助类关系图

6.4.1　几何变换(Transform)

Revit® 中的几何变换限于 3×4 的矩阵变换，一般用来对一个对象的位置从一个模型空间到另一个模型空间或坐标系空间的转换。变换矩阵是从模型空间的位置和方向构建出来的。三个向量(BasisX，BasisY 和 BasisZ 属性)和原点提供了所有的变换信息。矩阵构成如图 6-3 所示。

$$\begin{pmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} & \text{Origin.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} & \text{Origin.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} & \text{Origin.Z} \end{pmatrix}$$

图 6-3　变换矩阵

该矩阵在点坐标变换的应用如图 6-4 所示。

$$\begin{pmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} & \text{Origin.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} & \text{Origin.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} & \text{Origin.Z} \end{pmatrix} \times \begin{pmatrix} \text{XYZ.X} \\ \text{XYZ.Y} \\ \text{XYZ.Z} \\ 1 \end{pmatrix}$$

图 6-4　点坐标变换

Transform 类的 OfPoint(XYZ)方法就实现了上述的变换功能。下面的例子中，代码片段 6-7 也实现了同样的变换过程。

代码片段 6-7：几何

```
public static XYZ TransformPoint(XYZ point, Transform transform)
{
double x = point.X;
double y = point.Y;
double z = point.Z;
//获取变换的原点和基向量
XYZ b0 = transform.get_Basis(0);
XYZ b1 = transform.get_Basis(1);
XYZ b2 = transform.get_Basis(2);
XYZ origin = transform.Origin;
//对原来坐标系统的点在新的坐标系统进行变换
double xTemp = x * b0.X + y * b1.X + z * b2.X + origin.X;
double yTemp = x * b0.Y + y * b1.Y + z * b2.Y + origin.Y;
double zTemp = x * b0.Z + y * b1.Z + z * b2.Z + origin.Z;
return new XYZ(xTemp, yTemp, zTemp);
}
```

下面列举 Transform 类中常用的一些属性和方法。

1）恒等变换(Identity)

使用它不会改变一点或者一个向量，见图 6-5。

2）镜面反射(CreateReflection)

通过一个指定的平面做反射转换，见图 6-6。

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

图 6-5　恒等变换

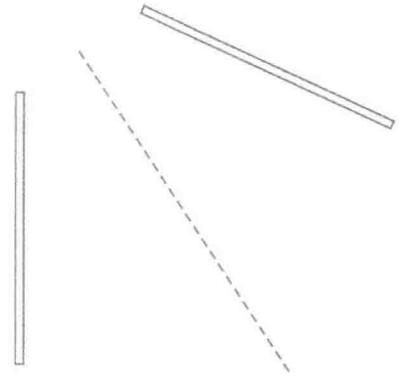


图 6-6　反射

如图 6-6 所示，对一堵墙通过一个引用平面进行镜像，可以使用 CreateReflection 方法通过指定平面得到相应的镜面反射变换。见代码片段 6-8。

代码片段 6-8：几何

```
// ReferencePlane refPlane =  GetRefPlane( );
    Transform mirTrans =  Transform.CreateReflection(refPlane.Plane);
```

3）旋转变换(CreateRotation)

通过点和轴进行角度旋转变换。Revit®提供了两个对应的方法：

① CreationRotation 方法创建了一个在原点(0，0，0)和指定的轴进行的角度变换。

② CreationRotationAtPoint 方法创建了一个在指定点和指定的轴进行的角度变换。

4）平移(CreateTranslation)

通过指定的向量进行平移转换。给定一个向量(x，y，z)，对应的平移向量如图 6-7 所示。

$$\begin{pmatrix} x & y & z \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \end{pmatrix}$$

图 6-7　平移

5）行列式(Determinant)

变换矩阵对应的行列式。几何上的意义是当进行线性变换时影响体积的比例因子。见图 6-8。

$$\begin{vmatrix} \text{BasisX.X} & \text{BasisY.X} & \text{BasisZ.X} \\ \text{BasisX.Y} & \text{BasisY.Y} & \text{BasisZ.Y} \\ \text{BasisX.Z} & \text{BasisY.Z} & \text{BasisZ.Z} \end{vmatrix}$$

图 6-8　行列式

6) 镜面反射变换(HasReflection)

HasReflection 的属性值是一个布尔值,表示该变换是否有一个镜面反射变换。镜面反射会改变空间的定向。

7) 缩放属性(Scale)

表示缩放变换的数值。

8) 逆变换(Inverse)

如果存在一个变换矩阵 ***B***,使得 $\boldsymbol{A} * \boldsymbol{B} = \boldsymbol{B} * \boldsymbol{A} = \boldsymbol{I}$ (identity),那么矩阵 ***A*** 是可逆的。***B*** 是 ***A*** 的逆矩阵。

9) IsIdentity

IsIdentity 的属性值是个布尔值,表示该变换是否是一个恒等变换。使用恒等变换不会改变一点或者一个向量。

10) IsTranslation

IsTranslation 的属性值是一个布尔值,表示该变换是否是一个平移。

11) Multiply

Multiply 的属性值是本变换和给定的变换的乘积。也可以用操作符 * 来对两个指定的变换进行乘法运算。

12) ScaleBasis

对该变换的基向量进行缩放,其返回值是缩放后的变换。

13) ScaleBasisAndOrigin

对该变换的基向量和原点进行缩放,其返回值是缩放后的变换。

14) OfPoint

对一点进行变换,其返回值是变换后的一点。

15) OfVector

对一向量进行变换,其返回值是变换后的一向量。

16) AlmostEqual

判断这个变换和给定的变换是否在容差(1.0e-09)内近似相等。

Revit® API 在几何类里提供了几个简便的进行变换的方法,常用的方法如表 6-2 所示。

表 6-2　　几何类进行变换的方法

类和对应方法	方法描述
Curve. CreateTransformed(Transform transform)	根据这个线型和给定的变换,创建一个新的线型实例
Instance. GetTransform()	得到这个实例的变换
Profile. get_Transformed(Transform transform)	根据这个轮廓和给定的变换,得到一个经过转换的新的轮廓
Mesh. get_Transformed(Transform transform)	根据这个网格和给定的变换,得到一个经过变换的新的网格

6.4.2　几何引用(Reference)

引用类 Reference 在元素创建的时候很有用。如创建尺寸时需要对应的引用。API 常使用的有四种引用类型,使用方法如下:

① 线上的端点引用：Curve. GetEndPointReference()方法。用来取一条线上起点和终点的引用，其中，参数 0 对应起点，参数 1 对应终点。

② 线引用：Curve. Reference 属性。用来获取该线的引用。

③ 面引用：Face. Reference 属性。用来获取该面的引用。

④ 边引用：Edge. Reference 属性。用来获取边的引用。

不同的引用类型不能混淆使用，需注意如下几点：

- NewLineBoundaryConditions 方法需要一个线的几何引用，如梁或墙的线引用。
- NewAreaBoundaryConditions 方法需要一个面引用，如墙或地板的面引用。
- NewPointBoundaryConditions 方法需要一个点引用，如梁或者柱子分析线的端点引用。

6.4.3 几何选项(Options)

几何选项定义分析几何生成模型的用户参数选项。它提供下面几种可供设置的用户选项：

- ComputeReferences：表示是否计算几何引用。
- DetailLevel：表示几何信息获取的详细程度。
- View：获取视图相关的元素几何信息。如果设置了相关的视图，那么详细程度就会使用视图的详细程度，即由视图来决定几何信息的详细程度。
- IncludeNonVisibleObjects：是否获取不可见的元素的几何对象信息，默认为否。

1) 选项 ComputeReferences

一般在需要获取几何引用时，需要先打开这个选项。如果设置为 false，Revit® 不会计算几何引用，那么在几何对象获取到的引用将返回空引用 null。

2) 选项 DetailLevel

DetailLevel 属性是 ViewDetailLevel 的枚举值，API 定义了三个枚举值来表示视图的详细程度。其中，枚举值 Fine 对应于 UI 上的精细，Medium 对应于中等，Coarse 对应于粗略。UI 界面的详细程度如图 6-9 所示。

不同的几何信息是根据不同详细程度设置属性获取的。例如，从 Revit® 中的一根梁获取到不同详细程度的几何信息再重新画出几何图形来，得到的结果如图 6-10 所示。



图 6-9 详细程度选项图

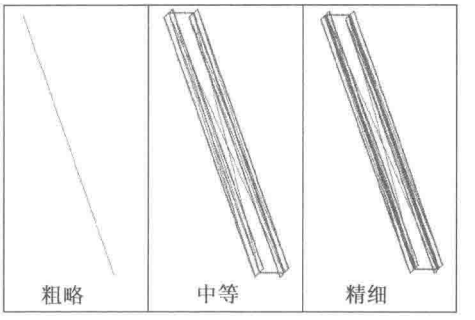


图 6-10 不同详细程度的几何模型图

3) 选项 View

如果设置不同的视图选项属性，获取到的几何信息也会不同。

在 Revit® 的 3D 视图中画一段楼梯，然后在 3D 视图属性中勾选裁剪视图、裁剪区域可见和剖面框三个参数。在裁剪区域中，调整剖面框到只显示楼梯的一部分。这时候，如果用 API 设置 Options. View 属性为 3Dview 并且取这段楼梯的几何信息，那么只有在裁剪区域中可见的那部分几何信息可以获取到。图 6-11 的左半部分显示了在 Revit® 中显示的上面所描述的楼梯，右半部分则显示了该段楼梯重新用 API 画出来的取到的几何信息。

如图 6-12 所示，左半部分显示在 Revit® 中画一段楼梯，然后再创建一个楼梯的剖面图。如果用 API 设置 Options. View 属性为这个剖面图并且取这段楼梯的几何信息，那么只能取到在剖面图这部分的楼梯的几何信息。右半部分是 API 用取到的几何数据重新绘制的几何图形。

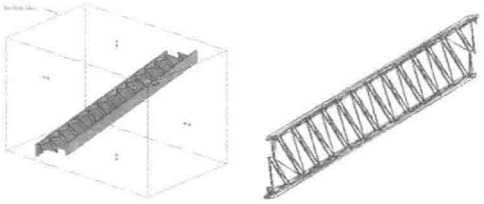


图 6-11　楼梯在 3D 视图的显示

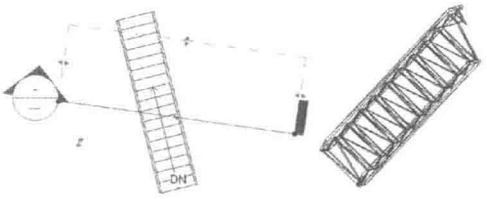


图 6-12　楼梯在剖面图的显示

4）几何选项创建

几何选项对象是用 Creation. Application 类的 NewGeometryOptions 方法来创建的，在对象创建之后，再设置相应的属性值。见代码片段 6-9。

代码片段 6-9：几何

```
Autodesk.Revit.DB.Options option =  this.Application.Create.NewGeometryOptions( );
option.ComputeReferences =  true;
option.DetailLevel =  ViewDetailLevel.Medium;
```

6.4.4　BoundingBoxXYZ

BoundingBoxXYZ 定义了一个三维的平行于坐标轴的长方体盒。它与 BoundingBoxUV 不同的是继承于 APIObject 对象。它的 Transform 属性可以用来将数据从局部坐标系变换到模型系统的坐标。如果要得到模型空间的长方体盒边界数据，可以用 Transform 依次转换各个数据。下面的小节显示了如何使用 BoundigBoxXYZ。

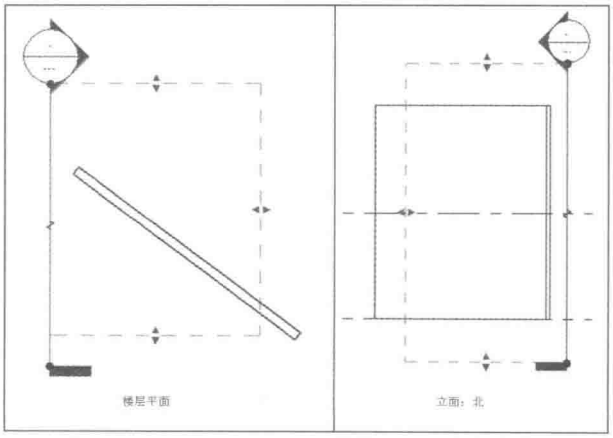


图 6-13　视图的边界

1）定义视图的边界

BoundingBoxXYZ 可以通过视图的裁剪盒属性（View. CropBox）来定义视图的边界。图 6-13 用一个剖视图来演示 Revit® 如何使用 Bounding

BoxXYZ 的。图 6-13 中的虚线表示了用一个 BoundingBoxXYZ 实例定义的视图裁剪盒属性，即该剖视图的边界。图 6-14 是对应的剖视图，剖视图中在边界外面的墙部分是不可见的，在裁剪盒里面的墙可见。

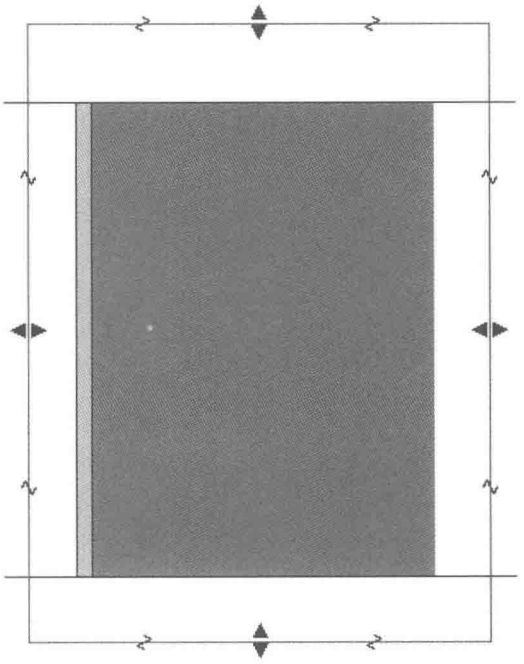


图 6-14　裁剪盒里的墙

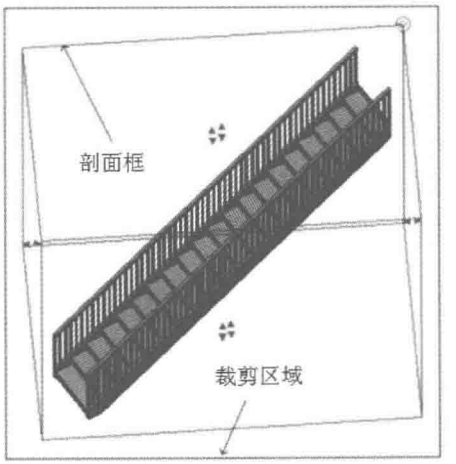


图 6-15　剖面框

2）定义三维视图剖面框

BoundingBoxXYZ 也可以通过三维视图的 GetSectionBox 方法来获取三维视图的剖面框(Section Box)。在 UI 界面的三维视图的属性框里，勾选剖面框属性，即可在视图中显示出来，如图 6-15 所示。

3）其他用法

① 定义元素的几何边界盒(Element. BoundingBox 属性)，这种情况 BoundingBoxXYZ 是平行于坐标轴的。

② 用于 ViewSection 类的 CreateSection 和 CreateDetail 方法。表 6-3 是这个类的主要属性。

表 6-3　　ViewSectiono 类的主要属性表

属性名	描述
Max/Min	最大/最小坐标值。这两个属性确定了平行于坐标的三维边界盒。最大值点是右上前点；最小值点是左下后点
Transform	从盒子坐标空间到模型空间的变换
Enable	表明边界盒是否打开

续表

属性名	描述
MaxEnabled/ MinEnabled	表明是否最大/最小边界在某一维度上是可用的，使用 0 参数代表 X 轴，1 代表 Y 轴，2 代表 Z 轴。 • 如果是一般的视图（ViewSection，ViewPlan 等），BoundingBoxXYZ 是从视图的 CropBox 属性获取的，那么表明是否可以用视图的裁剪盒来剪辑元素 • 如果 Enable 属性是 false，那么这两个属性也会返回 false • 如果视图的裁剪视图属性是打开的，即 API 中 View. CropBoxActive 为 true，那么 MaxEnalbed 和 MinEnalbed 属性都会返回 true • 如果视图的裁剪视图属性是关闭的，即 API 中 View. CropBoxActive 为 false，那么 MaxEnalbed 和 MinEnalbed 属性都会返回 false • 如果 BoundingBoxXYZ 是从三维视图的 GetSectionBox 方法获取的，那么返回值取决于三维视图的“剖面框”属性有没有勾选。如果勾选了该属性，即 API 中 BoundingBox. Enalbe 为 true，那么这两个属性都返回 true；否则返回 false • 如果 BoundingBoxXYZ 是从元素 Element. BoundingBox 属性获取的，那么这两个属性都返回 true
Bounds	通过索引获取最大或者最小边界值。用 0 参数代表最小值，1 代表最大值
BoundEnabled	通过索引获取边界是否打开。第一个参数为边界值，0 代表最小值，1 代表最大值；第二个参数是维度，0 代表 X 轴，1 代表 Y 轴，2 代表 Z 轴。如果要单独打开各个维度边界，必须整个边界盒是打开的

代码片段 6-10 演示了如何通过旋转 BoundingBoxXYZ 来修改三维视图的截面盒的。

代码片段 6-10：几何

```
public void ModifySectionBox( )
    {
        Document doc =  this.ActiveUIDocument.Document;
        using(Transaction transaction =  new Transaction(doc, "Modify Section Box"))
        {
            transaction.Start( );

            View3D view3d = doc.GetElement(new ElementId(186350)) as View3D;
            BoundingBoxXYZ box =  view3d.GetSectionBox( );
            if (false = =  box.Enabled)
            {
                TaskDialog.Show("Error", "The section box for View3D isn't Enable.");
                return;
            }
            //创建旋转变换
            XYZ origin =  new XYZ(0, 0, 0);
            XYZ axis =  new XYZ(0, 0, 1);
            Transform rotate =  Transform.CreateRotationAtPoint(axis, 2, origin);
            // 把旋转变换应用于三维视图的剖面框
            box.Transform =  box.Transform.Multiply(rotate);
            view3d.SetSectionBox(box);

            transaction.Commit( );
        }
    }
```

6.4.5 BoundingBoxUV

BoundingBoxUV 是平行于坐标轴的二维长方形。它有两个数据：最大坐标和最小坐标。这两个坐标共同定义了 BoundingBoxUV 的边界，见图 6-16。在图纸空间视图中，BoundingBoxUV 可以从视图的 Outline 属性获取其边界，见图 6-17。

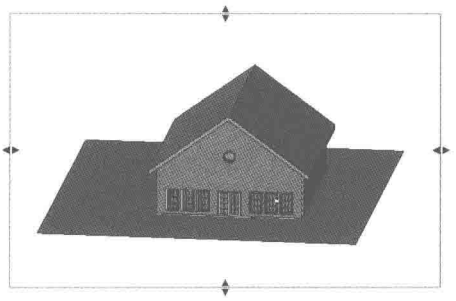


图 6-16 BoundingBoxUV 的边界

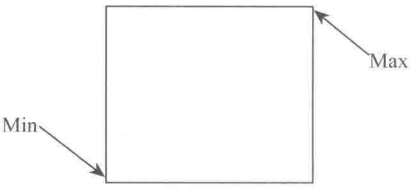


图 6-17 两个坐标

BoundingBoxUV 的两个坐标值：

- 最小值点：长方形的左下端点。
- 最大值点：长方形的右上端点。

代码片段 6-11 演示从三维视图中获取 Outline 属性，最大值点和最小值点。

代码片段 6-11：几何

```
Document projectDoc =  ActiveUIDocument.Document;
    View3D view =  projectDoc.ActiveView as View3D;
    if(view ! =  null)
    {
        BoundingBoxUV bbBoxUV =  view.Outline;
        UV max =  bbBoxUV.Max;
        UV min =  bbBoxUV.Min;
    }
```

注意：BoundingBoxUV 不能是倾斜的长方形，图 6-18 中的长方形就不可能是 Revit® 中的 BoundingBoxUV。

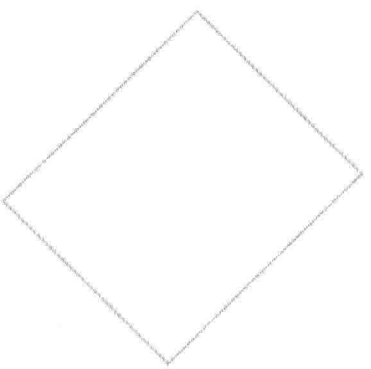


图 6-18 倾斜的长方形

6.5　几何集合类

API 提供了下面一些集合类用来储存或者遍历几何数据，所有这些类的方法和属性是类似的。

- 线：CurveArray，CurveArrayIterator
- 边：EdgeArray，EdgeArrayIterator，EdgeArrayArray，EdgeArrayArrayIterator
- 面：FaceArray，FaceArrayIterator
- 几何对象：GeometryObjectArray，GeometryObjectArrayIterator
- 实例：InstanceArray，InstanceArrayIterator
- 网格：MeshArray，MeshArrayIterator
- 引用：ReferenceArray，ReferenceArrayIterator
- 实体：SolidArray，SolidArrayIterator
- Double 值：DoubleArray，DoubleArrayIterator

6.6　实战：获取一根梁的几何数据

代码片段 6-12 演示了如何从一根梁中获取几何实体和线，也可以用类似的步骤来获取柱子或者支撑的几何数据。

代码片段 6-12：几何

```
//[in]geomElement, [out]curves, [out]solids
     public void GetCurvesFromABeam (FamilyInstance beam, Options options, CurveArray
     curves, SolidArray solids)
    {
        Autodesk.Revit.DB.GeometryElement geomElement = beam.get_Geometry
        (options);
        //找到所有的实体 solids 和线 curves
        AddCurvesAndSolids(geomElement, curves, solids);
    }

      private void AddCurvesAndSolids (GeometryElement geomElem, CurveArray curves,
      SolidArray solids)
    {
        foreach (GeometryObject geomObject in geomElem)
        {
            Curve curve = geomObject as Curve;
            if (null != curve)
            {
                curves.Append(curve);
                continue;
            }
            Solid solid = geomObject as Solid;
            if (null != solid)
            {
                solids.Append(solid);
                continue;
```

```
            }
            //如果 GeometryObject 是几何实例,则进行二次遍历
            GeometryInstance geomInst =  geomObject  as GeometryInstance;
            if (null ! =  geomInst)
            {
                  AddCurvesAndSolids (geomInst. GetSymbolGeometry (geomInst. Transform),
                  curves, solids);
            }
      }
}
```

第 7 章　族(Family)

7.1　族简介

族是一个包含通用属性(称作参数)集和相关图形表示的图元组。属于一个族的不同图元的部分或全部参数可能有不同的值,但是参数(其名称与含义)的集合是相同的。族中的这些变体称作族类型或类型。族是一个很重要的概念,很多图元都是使用族来定义的。如:家具(如桌子、椅子和橱柜)、结构柱(预制混凝土柱、角柱和其他柱)。

同一个族都具有相关的图形表示和一组相同的参数(族类型参数)。族使用族文档进行创建和编辑的,而系统族是不能够使用族文档进行编辑的(如墙和地板)。

在 API 中,提供了一系列的类和方法来提供管理族文档,包括族类型和族参数的管理、几何图元的管理、几何图元的可见性管理、族文档和工程文档交互(编辑族和载入族)等。

7.2　相关的主要 API 类

1) 族类型和族参数的管理

- FamilyManager:一个管理类,用于管理所有族类型相关的编辑行为。
- FamilyType:族类型,如图 7-1 中的"0762×0762 mm"就是一个族类型。
- FamilyParameter:族参数,一个族可以定义一系列的参数,而这些参数可以是属于族的类型或者族的实例。当参数属于族的类型时,同一类型的所有实例共享相同的参数值,而当参数属于族的实例时,每个实例可以有自己的参数值。

2) 几何图元的管理

FamilyItemFactory:一个专门用于族文件中的创建类,本章会列举说明其中几个重要的方法。

3) 几何图元的可见性管理

FamilyElementVisibility:用于编辑图元的可见性。

7.3　族类型和族参数的管理

7.3.1　获取管理类(FamilyManager)

FamilyManager 类代表了编辑族类型的最重要的工具,它与族类型类(FamilyType)和族参数类(FamilyParameter)一起实现了"族类型"对话框中的添加、修改和删除的一整套功能(图 7-1)。包括:族类型的创建和修改、族参数的创建和修改、族参数的关联约束等。

该类只存在于族文档中,在项目文档中,该类的实例是不能得到,而在族文档中,必然有一

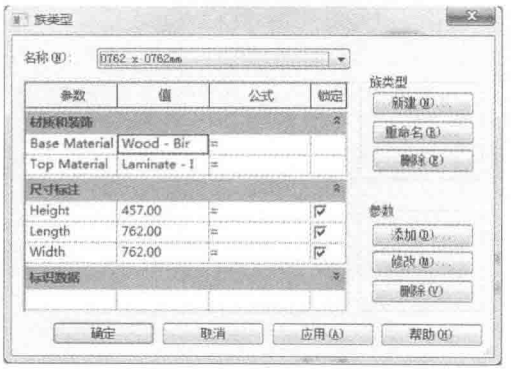


图 7-1　族类型

个该类的实例。代码片段 7-1 示例了如何从文档中得到该类的实例。

代码片段 7-1:获取族管理器

```
Documentdoc;  //得到族文档
if(doc.IsFamilyDocument)
    {
        //只有当 IsFamilyDocument 等于 true, FamilyManager 才能得到
FamilyManagerfamilyMgr= doc.FamilyManager;
    }
```

7.3.2 创建和编辑族类型

族类型是族的(尺寸)类型,每个族类型都有一组属性(族参数),包括带标签的尺寸标注及其值。也可以为族的标准参数(如材质、模型、制造商、类型标记等)添加值。

在族类型编辑器中,只能有一个族类型是当前类型,绝大多数的的修改(包括族类型和族参数)都只能针对当前族类型的。所以当打开一个族文档或者用族模板新创建一个族文档时候,该族文档也许还没有任何族类型,此时,应该先创建一个族类型来保证其他许多 API 调用的正常运行。

当前族类型可以从族管理类上面直接得到,而当前族类型可能为空,此时用户可以创建并设置一个族类型为当前类型。函数原型为:FamilyType CurrentType {get; set;}。

代码片段 7-2 示例了如何获取当前族类型。

代码片段 7-2:获取当前族类型

```
//得到 FamilyManager
FamilyManager familyMgr =  doc.FamilyManager;
FamilyType currentType =  familyMgr.CurrentType;
if(currentType ! =  null)
{
    //当前族类型可能为空,建议在其他操作前加上空检查
}
```

7.3.3 族类型的创建

使用 FamilyManager 中的 NewType 来实现族类型的添加,在创建完成后,Revit® 会把新创建的族类型设置为当前族类型。函数原型为:FamilyManager. NewType (string typeName)。

代码片段 7-3 示例了如何创建新的族类型。

代码片段 7-3:创建新的类型

```
//得到 FamilyManager
FamilyManager familyMgr =  doc.FamilyManager;
//族类型的名字,保证在所有族类型中唯一
string newTypeName =  "UniqueName";
//创建,得到 FamilyType 的实例
FamilyType famType =  familyMgr.NewType(newTypeName);
```

1) 族类型的删除

使用 DeleteCurrentType 来实现族类型的删除，一般而言，当函数把当前族类型删除后，第一个族类型会成为当前族类型，最后一个族类型是不能删除的。函数原型为：FamilyManager. DeleteCurrentType()。

代码片段 7-4 示例了如何删除族类型。

代码片段 7-4 删除族类型

```
//得到 FamilyManager
FamilyManager familyMgr = doc.FamilyManager;
if(familyMgr.CurrentType != null)
{
    //只有当前族类型存在，我们才能调用下面的删除方法
    familyMgr.DeleteCurrentType();
    //一般来说，当删除结束后，第一个族类型会成为当前类型，
    //但是为了确保安全，建议你显式设置成你需要的类型
    if (familyMgr.Types.Size != 0)
    {
        FamilyType type = familyMgr.Types.Cast< FamilyType> ().ElementAt(0);
        familyMgr.CurrentType = type;
    }
}
```

2) 族类型的重命名

使用 RenameCurrentType 方法可以实现族类型的重命名，如代码片段 7-5 所示。

代码片段 7-5：族类型重命名

```
//得到 FamilyManager
FamilyManager familyMgr = doc.FamilyManager;
//族类型的名字，保证在所有族类型中唯一
string newTypeName = "UniqueName";
familyMgr.RenameCurrentType(newTypeName);
```

3) 查看族类型的信息

从族管理类的属性(Types)可以很容易地得到族文档中所有已经存在的族类型，每个族类型通过 FamilyType 类来实现访问，从中可以得到族类型的名字和与之关联的族参数的值。如保得到族参数的值，会在“7. 3. 4 创建和编辑族参数”详细介绍。

7.3.4　创建和编辑族参数

族参数(FamilyParameter)是族文档中的核心，也是 Revit® 的一个核心概念，Revit® 的参数化驱动在族参数中有了相当的表现。本书立足于 API 的使用，不会涉及到很具体的概念，请参考 Revit® 的帮助文档。

1) 族参数的创建

为了便于理解，把族参数按照以下两种维度分类(图 7-2)：

- 按照族参数的类型；
- 按照族参数的作用范围。

(1) 按照族参数的类型

按照族参数的类型，可分为 3 类：

① 共享族参数：使用 FamilyManager 类中的 AddParameter(ExternalDefinition familyDefinition, BuiltInParameterGroup parameterGroup, bool isInstance)来创建，使用用法和一般族参数类似。

② 一般族参数：使用 FamilyManager 类中的 AddParameter(string parameterName, BuiltInParameterGroup parameterGroup, ParameterType parameterType, bool isInstance)来创建。此类族参数都有一个 Revit® 预定义的参数类型(如长度、体积等)，当这个参数类型确定以后，参数可以实现的操作也就确定了，例如，参数的单位(长度参数的单位可以是 cm，而体积的单位可以是 cm^3)，哪些参数可以关联(显然长度和体积的参数是不能关联到一起的)。

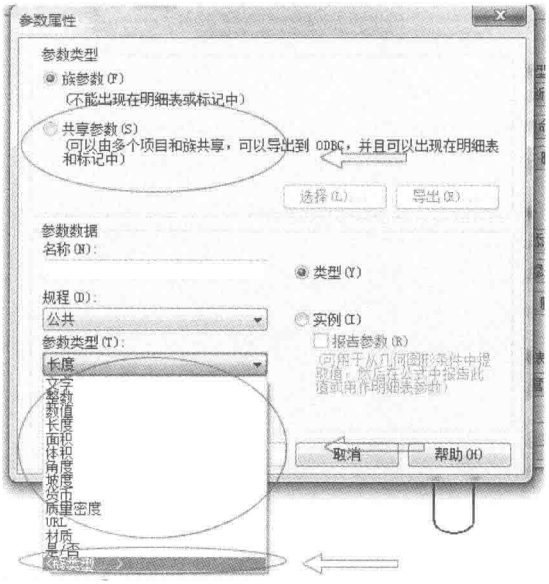


图 7-2 族参数

③ 族类型参数：使用 Familymanager 类中的 AddParameter(string parameterName, BuiltInParameterGroup parameterGroup, Category familyCategory, bool isInstance)来创建。此类族参数使用一个 Revit® 预定义的类别(Category)，只有该类型下的族类型可以设置为此参数的值。此类族参数是用来支持一个族文档嵌套到另一个族文档的情况的。

如图 7-2 所示，ExternalDefinition 类型输入参数对应了共享参数的定义，BuiltInParameterGroup 类型输入参数对应了参数分组方式，ParameterType 类型输入参数对应了参数类型中除了“<族类型...>”以外的所有选项。而 Category 类型输入参数对应了“<族类型...>”中的所有选项。

(2) 按照族参数的作用范围

- 类型参数：该参数绑定于一个族的类型。
- 实例参数：该参数绑定于一个族的实例。

注意：当族文档载入到工程文档以后，族的类型对应了 API 中的 FamilySymbol 类，而族的实例对应于 API 中的 FamilyInstance 类。

在 FamilyManager 类的 AddParameter 函数中，最后一个参数就是用来指定这个创建的族参数是实例参数还是类型参数。

2) 创建族参数的代码示例

(1) 添加共享族参数

这里给出使用和创建共享族参数的代码段(代码片段 7-6)，包括使用 API 创建共享参数的代码段。这与在工程文件中创建共享参数非常类似。

代码片段 7-6:创建共享族参数

```
Autodesk.Revit.ApplicationServices.Application app; //得到 Application
//得到 FamilyManager
FamilyManager familyMgr = doc.FamilyManager;

//共享参数的基本信息,包括定义文件路径,参数分组名称,参数名称和参数类型。
string sharedParameterFilePath = @"C:\SharedParameter.txt";
string sharedParameterGroupName = "Shared_Group";
string sharedParameter = "Shared_Parameter";
ParameterType sharedParameterType = ParameterType.Length;

//打开或创建共享参数定义文件。
app.SharedParametersFilename = sharedParameterFilePath;
DefinitionFile sharedDefinitonFile = app.OpenSharedParameterFile();
if (sharedDefinitonFile = = null)
  return;

//查找共享参数的分组名称,如果没找到,就创建一个。
DefinitionGroup sharedGroup = null;
sharedGroup = sharedDefinitonFile.Groups.get_Item(sharedParameterGroupName);
if (null = = sharedGroup)
  sharedGroup = sharedDefinitonFile.Groups.Create(sharedParameterGroupName);

//查找共享参数的定义,如果没有找到,就用名字和类型创建一个。
ExternalDefinition parameterDef = sharedGroup.Definitions.get_Item(sharedParameter) as
  ExternalDefinition;
if (null = = parameterDef)
  parameterDef = sharedGroup.Definitions.Create(sharedParameter, sharedParameterType)
  as ExternalDefinition;

//创建共享族参数
FamilyParameter newParameter = familyMgr.AddParameter(parameterDef,
  BuiltInParameterGroup.PG_CONSTRAINTS, true);
```

(2) 添加一般族参数

代码片段 7-7 示例了如何创建一般族参数。

代码片段 7-7:创建一般族参数

```
//得到 FamilyManager
FamilyManager familyMgr = doc.FamilyManager;
string paraName = "UniqueName"; //唯一
//设置族参数的类别和类型
BuiltInParameterGroup paraGroup = BuiltInParameterGroup.PG_LENGTH;
ParameterType paraType = ParameterType.Length;
//设置族参数为实例参数
bool isInstance = true;
//创建族参数
FamilyParameter newParameter = familyMgr.AddParameter(paraName, paraGroup, paraType,
isInstance);
```

(3) 添加族类型参数

代码片段 7-8 示例了如何创建族类型参数,以及该族参数的类型绑定到一个已加载的族的组上。

代码片段 7-8:创建族类型参数

```
//得到 FamilyManager
FamilyManager familyMgr = doc.FamilyManager;
string paraName = "族类型参数"; // 唯一
BuiltInParameterGroup paraGroup = BuiltInParameterGroup.PG_TEXT;
//设置族参数为实例参数
bool isInstance = true;

//得到一个已加载的族类型(FamilySymbol)的组
FilteredElementCollector cotr = new FilteredElementCollector(doc);
IList< Element> symbols = cotr.OfClass(typeof(FamilySymbol)).ToElements();
if (symbols.Count = = 0)
   return;  //未找到族类型,跳过创建
Category famCategory = symbols[0].Category;

//创建族参数
FamilyParameter newParameter = familyMgr.AddParameter(paraName, paraGroup, famCategory,
isInstance);
```

3) 查看族参数的信息

从族管理类的属性(Parameters)可以很容易地得到族文档中所有已经存在的族参数,每个族参数通过 FamilyParameter 类来实现访问其基本信息,包括参数的储存类型,显示类型,以及是否被公式驱动等一系列有用的信息。需要注意的是,FamilyParameter 的所有属性都是只读。

4) 族参数的重命名

和族类型的重命名类似,只需要调用 RenameParameter 函数把一个已创建的族参数设置一个新名字,显然新名字必须也是唯一的。函数原型:FamilyManager. RenameParameter (FamilyParameter familyParameter, string name)。

5) 修改族参数的作用范围

在创建族参数的函数中,最后一个参数设置了族参数的作用范围(类型参数或者实例参数)。该作用范围可以在族参数创建完成以后再进行修改。在族管理类里面提供了两种方法来实现这种修改。

Autodesk. Revit. DB. FamilyManager:

- void MakeType(FamilyParameter familyParameter)
- void MakeInstance(FamilyParameter familyParameter)

第一个函数用来把族参数转成类型参数,而第二个则是转成实例参数。

6) 报告参数

报告参数是一种参数类型,其值由族模型中的特定尺寸标注来确定。报告参数可从几何图形条件中提取值,然后使用它向公式报告数据或用作明细表参数。长度、半径、角度和

弧长度可用作报告参数，而其他大多数参数不能设置为报告参数(图 7-3)。

在 API 中，提供了两个函数来实现这个修改。

Autodesk. Revit. DB. FamilyManager：

• void MakeNonReporting (FamilyParameter familyParameter)；

• voidMakeReporting (FamilyParameter familyParameter)；

第一个函数用来把族参数取消设置为报告参数，而第二个则是设置成报告参数。

需要注意的是，这两个函数都是使用抛出异常来提示用户不能设置和取消报告参数。



图 7-3　报告参数

7) 设置族参数的公式

用户可以在族编辑器中使用公式来实现族参数之间的关联约束，如图 7-4 所示。

参数	值	公式	锁定
Duct Width	200.0 mm	=	☑
Diffuser Width	83.0	=	☑
Diffuser Length	1219.0	=	☑
其他			
Width 5	73.5	= Diffuser Half Width + 32 mm	☑
Width 4	10.4	= Diffuser Half Width / 4	☑
Width 3	13.8	= Diffuser Half Width / 3	☑
Width 2	20.8	= Diffuser Half Width / 2	☑
Width 1	54.0	= Diffuser Width / 2 + 12.5 mm	☑
Length 1	622.0	= Diffuser Length / 2 + 12.5 mm	☑
Diffuser Half Length	609.5	= Diffuser Length * 1 / 2	☑
Diffuser Half Width	41.5	= Diffuser Width * 1 / 2	☑
Slot Visibility 4	☐	= Slot Number = 4	

图 7-4　参数公式

需要注意的是，族参数之间的互相关联是有要求的，详见 Revit® 相关帮助文档。简单描述如下：

① 公式是需要能够计算的。

• 对于任意数据类型的参数，可以是一个可以计算出结果的加减乘除的计算式。

• 对于参数类型为字符串的参数，可以是一个字符串。

• 可以在计算式中加入另一个参数或者几个参数，实现参数驱动。

• 在计算式中，往往需要为一些数字加上单位。

② 如公式的表达式是不能计算出来的结果的，则不能用来作为公式。

• 公式是一个加减乘除的计算式，但是不符合计算式的运算规则，也就是说不能用这个计算式计算出一个结果的。

• 计算式中加入了一些族参数，而这些参数实际上并不存在的。

• 计算式中加入自己作为参数的，显然这个是死循环，是没法计算的。

③ 规则表达式计算出来的值，必须与相关联的族参数的单位一致。

• 一个表示面积的参数可以和另一个面积的参数关联，也可以和两个表示长度的乘积关联，但是不能直接和一个长度单位关联，因为它们的物理量是不同的。

• 当计算式中出现数字直接加减的时候，需要考虑这个数字的单位。

④ 当一个族参数定义了公式以后，它的值就可能被其他参数驱动，它的值可能也是只读的。

• 公式未引入其他参数，那么这个参数是只读的。

• 公式引入了一个其他参数，那么这两个参数应该是互为驱动的。这个参数是否为只读，由另一个参数是否只读决定。

• 公式引入了多个其他参数，那个这个参数被多个参数驱动，这个参数只能是只读的。

在 API 中，也可以设置族参数之间的相互关联，设置方法和 UI 上面类似，使用 FamilyManager. SetFormula(FamilyParameter familyParameter, string formula)指定一个符合规则的字符串给对应的族参数来设置关联规则。代码片段 7-9 示例了如何使用规则字符串来设置一个族参数的公式。

代码片段 7-9：设置族参数公式

```
/// < summary>
///这个方法演示如何创建族参数的公式。
///包括了如下几种类型的族参数：长度，面积，体积和字符串类型。
/// < /summary>
public void SetParameterFormula(
    FamilyManager familyMgr, FamilyParameter lengthPara,
    FamilyParameter areaPara, FamilyParameter volumePara,
    FamilyParameter commentPara)
{
    //字符串类型的族参数的公式，必须要带上""
    familyMgr.SetFormula(commentPara, "\"注释内容\"");

    //有单位的族参数(如长度类型)的公式，应该带上单位
    familyMgr.SetFormula(lengthPara, "1000mm");

    //族参数的公式需要符合其类型的定义
    //例如面积类型的参数可以是 2 个长度参数的乘积
    string lengtthParaName = lengthPara.Definition.Name;
    familyMgr.SetFormula(areaPara, lengtthParaName + " * " + lengtthParaName);

    //而体积类型的参数可以是 3 个长度参数的乘积
    familyMgr.SetFormula(volumePara, lengtthParaName
        + " * " + lengtthParaName + " * " + lengtthParaName);
}
```

8) 读写族参数的值

族参数的值是关联于族类型的，在每个族类型中都储存了各自类型的参数值，使用 API 可以实现读写族参数的值，读写方法略有不同，如下所示。

(1) 读取参数值

Autodesk. Revit. DB. FamilyType 类：

- double? AsDouble(FamilyParameter familyParameter)
- ElementId AsElementId(FamilyParameter familyParameter)
- int? AsInteger(FamilyParameter familyParameter)
- stringAsString(FamilyParameter familyParameter)
- stringAsValueString(FamilyParameter familyParameter)

和在工程文件里面读取参数的方式类似，族参数有四种存储类型，所以有四种对应的读取函

数;而 AsValueString 函数可以得到参数值的可视字符串。唯一不同的是,族参数是绑定到族类型的,需要先从族管理类中得到族类型和族参数,再来得到该参数在该类型上对应的值。

(2) 设置参数值

Autodesk. Revit. DB. FamilyManager 类:

- void Set(FamilyParameter familyParameter, double value)
- void Set(FamilyParameter familyParameter, ElementId value)
- void Set(FamilyParameter familyParameter, int value)
- void Set(FamilyParameter familyParameter, string value)

和读取族参数不同,设置族参数的方法在族管理类内,而需要的输入参数只有族参数和参数的值。前面提到,一个族参数的值对应于一个族类型的,所以该函数设置的值到当前族类型的,这个功能和 UI 上的一致。

读写族参数的要求和工程文档中的读写参数的要求相同。

注意:*在设置族参数的值以前,必须确保当前族类型是存在的。*

代码片段 7-10 示例了如何读取和设置族参数的值。

代码片段 7-10:设置族参数的值

```
public void SetFamilyParameterValue(
    FamilyManager familyMgr, FamilyParameter lengthPara,
    FamilyParameter commentPara, FamilyParameter IntPara,
    FamilyParameter materialPara, ElementId newMaterialId)
{
    //首先需要判断当前族类型是否存在,如果不存在,读写族参数都是不可行的
    FamilyType currentType = familyMgr.CurrentType;
    if (null == currentType)
        return;

    //读写长度类型的族参数;他的储存类型为 double
    double length = currentType.AsDouble(lengthPara).Value;
    familyMgr.Set(lengthPara, length + 1.25);

    //读写文字类型的族参数;他的储存类型为 string
    string comment = currentType.AsString(commentPara);
    familyMgr.Set(commentPara, "新的注释");

    //读写整数类型的族参数;他的储存类型为 int
    int integerVale = currentType.AsInteger(IntPara).Value;
    familyMgr.Set(IntPara, integerVale + 3);

    //读写材质类型的族参数;他的储存类型为 ElementId
    ElementId materialId = currentType.AsElementId(materialPara);
    familyMgr.Set(materialPara, newMaterialId);

    //您可以读写族参数的可视化文字表示的值
    string lengthText = currentType.AsValueString(lengthPara);
    familyMgr.SetValueString(lengthPara, "10m");
}
```

9）族参数的删除

使用族管理类的 RemoveParameter 函数来实现族类型的删除。但是内建的族参数（如“标识数据”分组下的“制造商”参数）是不能删除的。显然删除族参数以后，所有和该参数有关联的信息也会被清空，如任何使用了这个参数的公式等。

函数原型：

Autodesk. Revit. DB. FamilyManager 类：void RemoveParameter（FamilyParameter familyParameter）。

10）关联族参数

通过关联族参数，可以在项目视图中控制主体族中嵌套族的参数，可以控制实例参数或类型参数。类似地，也可以用来控制内建族（如实体拉伸）的实例参数。

若要关联参数，这些参数必须是相同的类型。例如，将主体族中的文字参数与嵌套族中的文字参数关联。

以实体拉伸的实例参数为例，如图 7-5 所示，选中一个实体拉伸的实例，在它的“属性”选项板，可以看见图 7-5 中标识的小按钮，点击这些小按钮实现实例参数和族参数的关联。

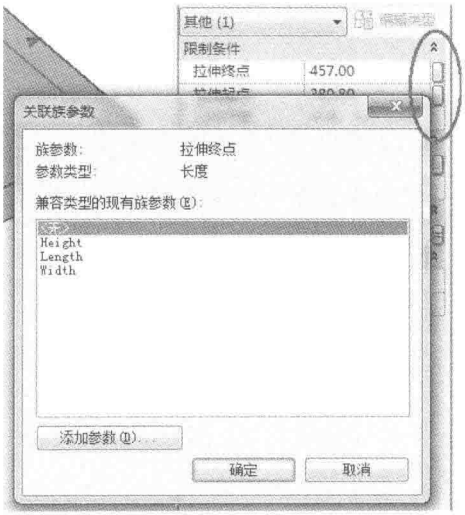


图 7-5　关联参数

在 API 中提供了两种方法来实现类似的功能。

Autodesk. Revit. DB. FamilyManager 类：

- void AssociateElementParameterToFamilyParameter（Parameter elementParameter，FamilyParameter familyParameter）
- boolCanElementParameterBeAssociated（Parameter elementParameter）

第一个函数用来实施关联操作，第二个函数用来判断实例（或者类型）参数是否可能和一个族参数关联的。FamilyParameter 类中专门提供了一个属性来给出该族参数已经绑定了的实例（或者类型）参数的集合。属性原型为：ParameterSet AssociatedParameters {get;}。

代码片段 7-11 示例了如何把关联族参数关联到实例参数。

代码片段 7-11：关联族参数

```
public void AssociateParametersInFamilyDocument(FamilyManager familyMgr,
    Parameter lengthElementPara, //长度类型的图元参数
    FamilyParameter lengthFamilyPara    //长度类型的族参数
    )
{
    ParameterType    lengthElementParaType    =    lengthElementPara.    Definition.
    ParameterType;
    ParameterType    lengthFamilyParaType    =    lengthFamilyPara.    Definition.
    ParameterType;
```

```
    //能关联起来的图元参数和族参数的类型要相同
    if (lengthElementParaType ! =  lengthFamilyParaType)
    {
        //注意:有些类型不同的图元参数和族参数也是可以关联的。
        //比如 PipeSize 和 Length 类型的参数可以关联,因为他们代表了相同的物理量:长度
        return;
    }

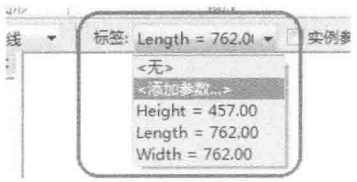
    //判断该长度单位是否能够和族参数关联
    if (! familyMgr.CanElementParameterBeAssociated(lengthElementPara))
        return;

    //进行关联操作
         familyMgr.   AssociateElementParameterToFamilyParameter   ( lengthElementPara,
         lengthFamilyPara);
}
```

11) 族参数和尺寸标注的关联

在族文档中可以使用尺寸标注的标签来关联一个族参数,在将族载入到项目中之后,可以在"属性"选项板上修改任何实例参数,或者打开"类型属性"对话框修改类型参数值,以此可以实现尺寸标注由参数驱动。

在 UI 上面,可以使用如下两种步骤中的任何一种来实现标签的关联:



方法一:
- 在族编辑器中,选择尺寸标注。
- 在选项栏上,选择一个参数或者选择"<添加参数...>"并创建一个参数作为"标签"。



方法二:
- 在族编辑器中,选择尺寸标注。
- 在属性对话框中,为"标签"属性选择一个族参数。

在 API 中,相同的功能可以使用 Dimension 类中的 FamilyLabel 属性进行设置,在设置后,同样实现尺寸标注被族参数驱动的功能,如代码片段 7-12 所示。

代码片段 7-12:关联族参数和尺寸标注

```
public void AssociateParameterToDimension(
    Dimension dimension, FamilyParameter familyPara)
```

```
{
    //获取当前关联的族参数
    FamilyParameter currentPara = dimension.FamilyLabel;
    if(null != currentPara)
    {
        //新关联的族参数类型应该和现有的族参数类型一致。
        //就像上个例子提到的一样,它们至少应该是物理意义相同。
            if (currentPara.Definition.ParameterType != familyPara.Definition.
            ParameterType)
            return;
    }

    //关联新的族参数
    dimension.FamilyLabel = familyPara;

    //清空关联
    dimension.FamilyLabel = null;
}
```

12）小结

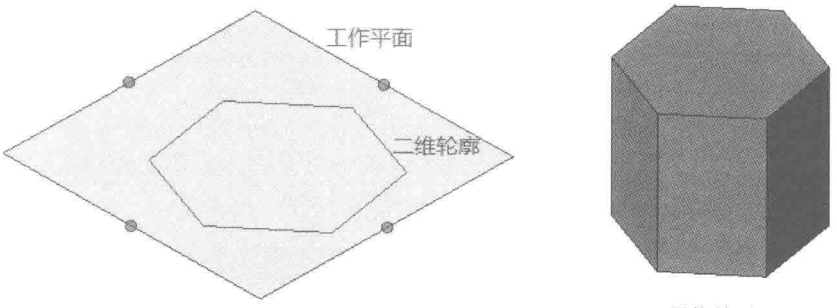
在族文档中，通过关联族参数来控制嵌套族的参数以及族参数和尺寸标注关联，实现了由族参数驱动族文档模型的方式。通过族参数的公式，可以把族参数互相关联起来。由此，整个族文档的信息可以被少数的几个族参数驱动了。

7.4 几何图元的管理

几何图元的创建是族文档的另一个重要部分，Revit® 支持实体（实心）几何图元和剪切体（空心）几何图元的创建，并通过各种图元的结合来创建各种复杂的几何形体。Revit® 支持完全相同的实体和剪切体的创建方式，共五种：拉伸，融合，旋转，放样和放样融合。本节将会一一介绍如何使用 API 来创建对应的图元。

7.4.1 拉伸

拉伸是最容易创建的形状。可以在工作平面上绘制形状的二维轮廓，然后拉伸该轮廓使其与绘制它的平面垂直。根据描述，拉伸操作需要三个输入：工作平面，工作平面上的二维轮廓（必须闭合）和拉伸的长度。在界面上，拉伸操作的大体步骤如图 7-6 所示。



工作平面和二维轮廓　　最终结果

图 7-6　拉伸

在 Revit® API 中,同样需要这三个输入来创建拉伸图元。创建拉伸图元的函数原型如下:

Autodesk. Revit. Creation. FamilyItemFactory 类: Extrusion NewExtrusion (bool isSolid, CurveArrArray profile, SketchPlane sketchPlane, double end)。

参数说明:

- isSolid:说明该拉伸是实体还是剪切体
- profile:一个平面上的二维轮廓(必须闭合)
- sketchPlane:工作平面
- end:拉伸长度

注意:

① 拉伸的方向由工作平面的法向量确定。

② API 中,并不要求二维轮廓一定要在 sketchPlane 工作平面上(大多数情况下,二维轮廓是在工作平面上的),它们之间必须满足的是二维轮廓所在在平面和工作平面一定要平行。

代码片段 7-13 示例了如何创建拉伸体。

代码片段 7-13:创建拉伸体

```
privateApplication m_revit; // 已获得 Revit 的 Application 的实例
privateDocument m_familyDocument;  // 已获得族文档的实例
//创建工作平面的函数,输入为平面的原点和法向量
internal SketchPlane CreateSketchPlane(Autodesk.Revit.DB.XYZ normal, Autodesk.Revit.DB.
XYZ origin)
{
//首先创建几何平面
   Plane geometryPlane =  m_revit.Create.NewPlane(normal, origin);
   if (null = =  geometryPlane)
   {
returnnull;
   }
   //根据几何平面创建工作平面
   SketchPlane plane =  SketchPlane.Create(m_familyDocument, geometryPlane);
   if (null = =  plane)
   {
returnnull;
   }
   return plane;
}

//创建用于拉伸的轮廓线
private CurveArrArray CreateExtrusionProfile()
{
//轮廓线可以包括一个或者多个闭合的轮廓,所以最后返回是 CurveArrArray
   CurveArrArray curveArrArray =  new CurveArrArray();
   CurveArray curveArray1 =  new CurveArray();

   //创建一个正方体的轮廓线,先创建点,再创建线,最后组合成轮廓。
```

```
    Autodesk.Revit.DB.XYZ p0 = Autodesk.Revit.DB.XYZ.Zero;
    Autodesk.Revit.DB.XYZ p1 = new Autodesk.Revit.DB.XYZ(10, 0, 0);
    Autodesk.Revit.DB.XYZ p2 = new Autodesk.Revit.DB.XYZ(10, 10, 0);
    Autodesk.Revit.DB.XYZ p3 = new Autodesk.Revit.DB.XYZ(0, 10, 0);
    Line line1 = Line.CreateBound(p0, p1);
    Line line2 = Line.CreateBound(p1, p2);
    Line line3 = Line.CreateBound(p2, p3);
    Line line4 = Line.CreateBound(p3, p0);
    curveArray1.Append(line1);
    curveArray1.Append(line2);
    curveArray1.Append(line3);
    curveArray1.Append(line4);
    curveArrArray.Append(curveArray1);
    return curveArrArray;
}

private void CreateExtrusion(FamilyItemFactory familyCreator)
{
    //调用函数创建拉伸的轮廓线和工作平面
    CurveArrArray curveArrArray = CreateExtrusionProfile();
    SketchPlane sketchPlane = CreateSketchPlane(XYZ.BasisZ, XYZ.Zero);

    //调用 API 创建拉伸(实心正方体)
    Extrusion rectExtrusion = familyCreator.NewExtrusion(true, curveArrArray, sketchPlane,
    10)
    //可能我们会希望把拉伸体移动到希望的位置上
    XYZ transPoint1 = new XYZ(- 16, 0, 0);
    ElementTransformUtils.MoveElement(m_familyDocument, rectExtrusion.Id, transPoint1);
}
```

1) 读写拉伸属性

在拉伸图形(Extrusion)中,可以通过属性访问到拉伸图形的各种信息。

① IsSolid 属性:该拉伸是实体还是剪切体。

② Sketch 属性:拉伸的草图,包括拉伸的二维轮廓线和工作平面。

③ StartOffset 和 EndOffset:拉伸相对于工作平面的起始和终止偏移。

2) 草图(Sketch 类)

在 Revit® 中,有一些图元是使用草图模式创建的,例如,工程文件中的楼板、天花板和族文件中的拉伸等。而草图模式是指 Revit® 中的一种环境,使用该环境可绘制其尺寸或形状不能自动确定的图元。Sketch 类就是用来访问它创建的草图,包括草图模式中绘制形状和工作平面。

Sketch 类有两个属性,且都是只读的:

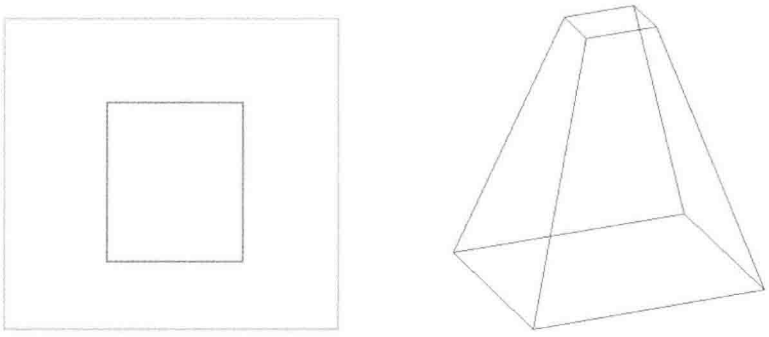
① Profile:用来获取草图中的轮廓线。

② SketchPlane:用来获取草图中的工作平面。

7.4.2 融合

融合可将两个轮廓(边界)融合在一起。例如,如果绘制一个大矩形,并在其顶部绘制一

个小矩形，则 Revit® 会将这两个形状融合在一起。根据描述，融合操作需要一个底部轮廓和一个顶部轮廓，工作平面和融合的深度，如图 7-7 所示。



(a) 融合底部和顶部轮廓　　(b) 完成的融合体

图 7-7　融合

在 Revit® API 中，同样需要类似的输入来创建融合图元。创建融合图元的函数原型如下：

Autodesk. Revit. Creation. FamilyItemFactory 类：

- Blend NewBlend(bool isSolid, CurveArray topProfile, CurveArray baseProfile, SketchPlane sketchPlane)

参数说明：

- isSolid：说明该融合是实体还是剪切体；
- topProfile：顶部的二维轮廓(必须闭合)；
- baseProfile：底部的二维轮廓(必须闭合)；
- sketchPlane：工作平面，通常底部轮廓在工作平面上。

注意：底部和顶部的轮廓必须是平行的；而二者之间的距离决定了融合的深度。

代码片段 7-14 示例了如何创建融合体。

代码片段 7-14：创建融合体

```
private void CreateBlend(FamilyItemFactory familyCreator)
{
    CurveArray topProfile =  new CurveArray();
    CurveArray baseProfile =  new CurveArray();

//创建工作平面，CreateSketchPlane 见上一个例子。
    Autodesk.Revit.DB.XYZ normal =  Autodesk.Revit.DB.XYZ.BasisZ;
    SketchPlane sketchPlane =  CreateSketchPlane(normal, Autodesk.Revit.DB.XYZ.Zero);

    //创建底部的闭合二维轮廓
    Autodesk.Revit.DB.XYZ p00 =  Autodesk.Revit.DB.XYZ.Zero;
    Autodesk.Revit.DB.XYZ p01 =  new Autodesk.Revit.DB.XYZ(10, 0, 0);
    Autodesk.Revit.DB.XYZ p02 =  new Autodesk.Revit.DB.XYZ(10, 10, 0);
    Autodesk.Revit.DB.XYZ p03 =  new Autodesk.Revit.DB.XYZ(0, 10, 0);
```

```
    Line line01 = Line.CreateBound(p00, p01);
    Line line02 = Line.CreateBound(p01, p02);
    Line line03 = Line.CreateBound(p02, p03);
    Line line04 = Line.CreateBound(p03, p00);
    baseProfile.Append(line01);
    baseProfile.Append(line02);
    baseProfile.Append(line03);
    baseProfile.Append(line04);

//创建顶部的闭合二维轮廓
    Autodesk.Revit.DB.XYZ p10 = new Autodesk.Revit.DB.XYZ(5, 2, 10);
    Autodesk.Revit.DB.XYZ p11 = new Autodesk.Revit.DB.XYZ(8, 5, 10);
    Autodesk.Revit.DB.XYZ p12 = new Autodesk.Revit.DB.XYZ(5, 8, 10);
    Autodesk.Revit.DB.XYZ p13 = new Autodesk.Revit.DB.XYZ(2, 5, 10);
    Line line11 = Line.CreateBound(p10, p11);
    Line line12 = Line.CreateBound(p11, p12);
    Line line13 = Line.CreateBound(p12, p13);
    Line line14 = Line.CreateBound(p13, p10);
    topProfile.Append(line11);
    topProfile.Append(line12);
    topProfile.Append(line13);
    topProfile.Append(line14);

    //利用底部和顶部的轮廓来创建融合
    Blend blend = familyCreator.NewBlend(true, topProfile, baseProfile, sketchPlane);
}
```

1）读写融合属性

和 Extrusion 类类似，可以通过 Blend 类来访问融合图形的各种信息。主要的不同之处有如下两点：

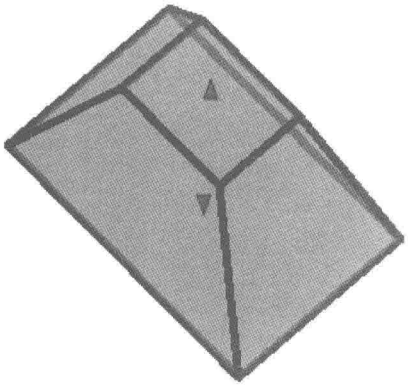
① 融合因为有底部和顶部两个轮廓线，所以 Blend 类有两个属性来分别获取底部和顶部的 Sketch 实例。需要注意的是：在创建函数（NewBlend）中，输入 CurveArray 来作为底部和顶部的轮廓线，而 Blend 类获取到的底部和顶部的轮廓线是 CurveArrArray 类型的。因为 Blend 的底部和顶部轮廓可以是多个闭合轮廓组成的复杂轮廓线；而在 Revit® 内部，一个闭合轮廓是一个 CurveArray，它会把创建函数输入的轮廓线分解成一个或多个闭合轮廓线的集合。

② 融合需要融合底部和顶部的轮廓线，所以定义了两者轮廓线上顶点的映射关系，API 提供了一套方法来读取和修改该映射关系，来达到如图 7-8 所示的效果。

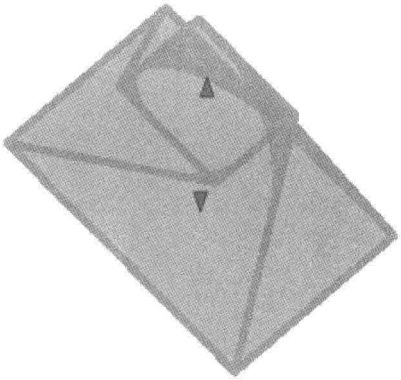
Autodesk. Revit. DB. Blend 类：

- VertexIndexPairArray GetVertexConnectionMap()
- void SetVertexConnectionMap(VertexIndexPairArray vertexMap)

代码片段 7-15 示例了如何编辑融合体的顶部和底部的映射关系来改变融合体。



(a) 修改前



(b) 修改后

图 7-8　修改融合

代码片段 7-15:编辑融合体

```
public void AccessBlend(Blend blend)
{
    //得到融合体的底部和顶部轮廓线
    CurveArrArray baseProfile =  blend.BottomProfile;
    CurveArrArray topProfile =  blend.TopProfile;

    //通过修改 TopOffset 属性来让融合体的顶部更高
    blend.TopOffset =  blend.TopOffset +  1;

    //修改融合体顶部和底部的映射关系
    VertexIndexPairArray currentMap =  blend.GetVertexConnectionMap();
    VertexIndexPairArray newMap =  new VertexIndexPairArray();
    List< int>  topIndexes =  new List< int> ();
    List< int>  bottomIndexes =  new List< int> ();
    foreach(VertexIndexPair pair in currentMap)
    {
        topIndexes.Add(pair.Top);
        bottomIndexes.Add(pair.Bottom);
    }
    int lastTopindex =  topIndexes.Last();
    topIndexes.RemoveAt(topIndexes.Count- 1);
    topIndexes.Insert(0, lastTopindex);
    for(int idx =  0; idx <  bottomIndexes.Count; idx+ + )
    {
        newMap.Append(new VertexIndexPair(topIndexes[idx], bottomIndexes[idx]));
    }
    blend.SetVertexConnectionMap(newMap);
}
```

7.4.3　旋转

旋转是指围绕轴旋转某个形状而创建的形状。可以旋转形状一周或不到一周。如果轴与旋转造型接触,则产生一个实心几何图形。根据描述,旋转操作需要一个闭合的轮廓线,

旋转轴，工作平面和旋转角度。如图 7-9 所示。

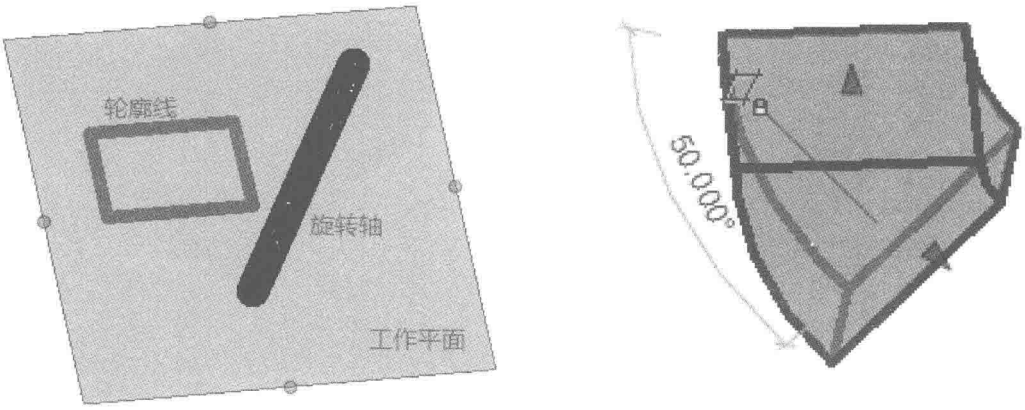


图 7-9　旋转

在 Revit® API 中，同样需要类似的输入来创建旋转图元。创建旋转图元的函数原型如下：

Autodesk. Revit. Creation. FamilyItemFactory 类：

• Revolution NewRevolution (bool isSolid, CurveArrArray profile, SketchPlane sketchPlane, Line axis, double startAngle, double endAngle)

参数说明：

• isSolid：说明该旋转体是实体还是剪切体；

• profile：旋转体的二维轮廓的集合（需要闭合），比如“回”字形的轮廓线包括了内外两个方形的轮廓线；

• sketchPlane：工作平面，并不要求轮廓线在工作平面上；

• Axis：旋转轴，必须和轮廓线在同一平面；

• startAngle：旋转的起始角度；

• endAngle：旋转的终止角度。

注意：*旋转的方向由工作平面的法向量确定。*

代码片段 7-16 示例了如何创建旋转体。

代码片段 7-16：创建旋转体

```
private void CreateRevolution(FamilyItemFactory familyCreator)
{
    CurveArrArray curveArrArray =  new CurveArrArray();
    CurveArray curveArray =  new CurveArray();

//创建工作平面，CreateSketchPlane 参见 Extrusion 中的例子。
    Autodesk.Revit.DB.XYZ normal =  Autodesk.Revit.DB.XYZ.BasisZ;
    SketchPlane sketchPlane =  CreateSketchPlane(normal, Autodesk.Revit.DB.XYZ.Zero);
```

```
    //创建一个正方体的轮廓线
    Autodesk.Revit.DB.XYZ p0 =  Autodesk.Revit.DB.XYZ.Zero;
    Autodesk.Revit.DB.XYZ p1 =  new Autodesk.Revit.DB.XYZ(10, 0, 0);
    Autodesk.Revit.DB.XYZ p2 =  new Autodesk.Revit.DB.XYZ(10, 10, 0);
    Autodesk.Revit.DB.XYZ p3 =  new Autodesk.Revit.DB.XYZ(0, 10, 0);
    Line line1 =  Line.CreateBound(p0, p1);
    Line line2 =  Line.CreateBound(p1, p2);
    Line line3 =  Line.CreateBound(p2, p3);
    Line line4 =  Line.CreateBound(p3, p0);
    curveArray.Append(line1);
    curveArray.Append(line2);
    curveArray.Append(line3);
    curveArray.Append(line4);

// NewRevolution 函数需要一个轮廓线的集合来满足复杂的轮廓线,如"回"字形。
curveArrArray.Append(curveArray);

//创建旋转轴
    Autodesk.Revit.DB.XYZ pp =  new Autodesk.Revit.DB.XYZ(1, - 1, 0);
    Line axis1 =  Line.CreateBound(Autodesk.Revit.DB.XYZ.Zero, pp);

    //利用轮廓线和旋转轴来创建旋转体
      Revolution  revolution1  =  familyCreator. NewRevolution ( true,  curveArrArray,
      sketchPlane, axis1, - Math.PI, 0);
}
```

和 Extrusion 类类似,可以通过 Revolution 类来访问旋转图形的各种信息,并能修改旋转的起始和终止角度。

7.4.4　放样

放样是用于创建需要绘制或应用轮廓(造型)并沿绘制或者选择的路径拉伸此轮廓形成形体。根据描述,放样需要两个数据:用于拉伸的轮廓和拉伸的路径。在 UI 中,轮廓可以是绘制产生的,也可以是由加载的轮廓族定义的;而路径也可以是拾取路径或者绘制路径。如图 7-10 所示。



图 7-10　放样

在 Revit® API 中,可以输入类似的信息来创建放样图元。创建放样图元的函数原型如下:

Autodesk. Revit. Creation. FamilyItemFactory 类:

① Sweep NewSweep(bool isSolid, ReferenceArray path, SweepProfile profile, int

profileLocationCurveIndex, ProfilePlaneLocation profilePlaneLocation);

② Sweep NewSweep(bool isSolid, CurveArray path, SketchPlane pathPlane, SweepProfile profile, int profileLocationCurveIndex, ProfilePlaneLocation profilePlaneLocation);

参数说明:

• isSolid:说明该放样是实体还是剪切体。

• path:放样体的拉伸路径,该路径可以是绘制的路径也可以是拾取的路径,所以放样的创建有两个重载函数:

○ 拾取路径:path 参数的类型是 ReferenceArray,代表拾取的边缘或者线的集合。

○ 绘制路径:path 参数的类型是 CurveArray,可以设定一个路径的平面和一个 2D 线段组成的路径来描述在指定平面上的一个路径。而路径平面由第二个重载函数中的 pathPlane 参数来指定。

• profile:放样需要的轮廓线类型是 SweepProfile,不同于之前的拉伸,旋转直接使用轮廓线类型,原因是放样的轮廓不仅可以是由绘制的轮廓线组成,也可以是由载入的轮廓族来定义。所以 SweepProfile 类有两个子类:

○ CurveLoopsProfile 类:由轮廓线组成的轮廓;可以通过 Autodesk. Revit. Creation. Application 类中的 NewCurveLoopsProfile 方法来把线段集合组成的轮廓线转化成这里需要的 SweepProfile 类型。

○ FamilySymbolProfile 类:由轮廓族类型来定义的轮廓;可以通过 Autodesk. Revit. Creation. Application 类中的 NewFamilySymbolProfile(FamilySymbol)来获取由轮廓族类型定义的轮廓。

• profileLocationCurveIndex 和 profilePlaneLocation:前面讲过,放样体是沿路径拉伸轮廓形成的形体,那么要拉伸的轮廓放到那么呢? 可已选择放到拉伸的起点或者终点,又或者是拉伸路径的其他位置。这两个参数是创建放样特有的,是用来定义轮廓的位置,可以把它放到拉伸路径的每一个线段的起点,终点或者中心点。

代码片段 7-17 示例了如何创建放样体。

代码片段 7-17:创建放样体

```
private void CreateSweep(Autodesk.Revit.Creation.Application appCreator,
    FamilyItemFactory familyCreator)
{
    CurveArrArray ararr =  new CurveArrArray();
    CurveArray arr =  new CurveArray();

    //创建路径平面,CreateSketchPlane 参见 Extrusion 中的例子。
    Autodesk.Revit.DB.XYZ normal =  Autodesk.Revit.DB.XYZ.BasisZ;
    SketchPlane sketchPlane =  CreateSketchPlane(normal, Autodesk.Revit.DB.XYZ.Zero);

//创建用于放样的轮廓,这里创建轮廓线再生成轮廓的方式。
    Autodesk.Revit.DB.XYZ pnt1 =  new Autodesk.Revit.DB.XYZ(0, 0, 0);
    Autodesk.Revit.DB.XYZ pnt2 =  new Autodesk.Revit.DB.XYZ(2, 0, 0);
```

```
    Autodesk.Revit.DB.XYZ pnt3 =  new Autodesk.Revit.DB.XYZ(1, 1, 0);
    arr.Append(Arc.Create(pnt2, 1.0d, 0.0d, 180.0d, Autodesk.Revit.DB.XYZ.BasisX,
    Autodesk.Revit.DB.XYZ.BasisY));
    arr.Append(Arc.Create(pnt1, pnt3, pnt2));
    arrarr.Append(arr);
    SweepProfile profile =  appCreator.NewCurveLoopsProfile(arrarr);

//创建用于放样的路径,该路径包括两条线段。
    Autodesk.Revit.DB.XYZ pnt4 =  new Autodesk.Revit.DB.XYZ(10, 0, 0);
    Autodesk.Revit.DB.XYZ pnt5 =  new Autodesk.Revit.DB.XYZ(0, 10, 0);
    Autodesk.Revit.DB.XYZ pnt6 =  new Autodesk.Revit.DB.XYZ(5, 13, 0);
    Curve curve =  Line.CreateBound(pnt4, pnt5);
    Curve curve1 =  Line.CreateBound(pnt5, pnt6);
    CurveArray curves =  new CurveArray();
    curves.Append(curve);
    curves.Append(curve1);

    //利用轮廓和拉伸路径来创建放样,轮廓线位于拉伸路径的第二条线段的中心点。
     Sweep sweep1 =  familyCreator.NewSweep(true, curves, sketchPlane, profile, 1,
    ProfilePlaneLocation.MidPoint);
}
```

7.4.5　放样融合

通过放样融合工具可以创建一个具有两个不同轮廓的融合体,然后沿某个路径对其进行放样。放样融合的造型由二维路径以及两个轮廓确定。放样融合是放样和融合两种方式的综合体。一方面它和放样有相似之处:轮廓可以是绘制的产生的,也可以是由加载的轮廓族定义的;路径可以是拾取路径或者绘制路径等。另一方面它和融合有相似之处:必须有两个轮廓(顶部和底部轮廓),而且这两个轮廓是必须位于放样路径的两端;可以修改顶部和底部的轮廓顶点的映射关系。特殊性:放样路径只能是一条线段或者一条图元的边(创建放样是允许路径由多条连续的线段或者边组成的)。如图 7-11 所示。

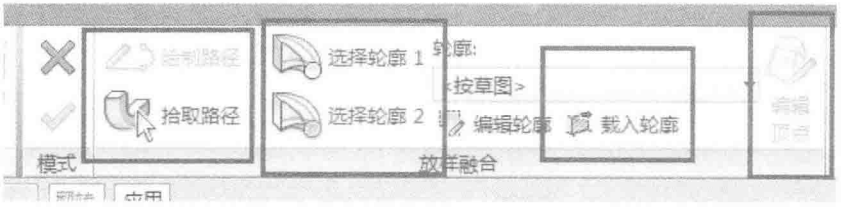


图 7-11　放样融合

在 Revit® API 中,可以输入类似的信息来创建放样融合图元;创建放样融合图元的函数原型如下:

Autodesk. Revit. Creation. FamilyItemFactory 类:

① SweptBlend NewSweptBlend (bool isSolid, Reference path, SweepProfile bottomProfile, SweepProfile topProfile);

② SweptBlend NewSweptBlend(bool isSolid, Curve path, SketchPlane pathPlane, SweepProfile bottomProfile, SweepProfile topProfile);

参数说明：

- isSolid：说明该放样融合是实体还是剪切体；
- path：和放样类似，不同的是这里只能有一条线段或者一条边；
- bottomProfile 和 topProfile：和放样的 profile 参数意义相同，详见 7.4.4 节中的参数说明。

代码片段 7-18 示例了如何创建放样融合体。

代码片段 7-18：创建放样融合体

```
private void CreateSweepBlend(Autodesk.Revit.Creation.Application appCreator,
    FamilyItemFactory familyCreator)
{
    //创建底部的轮廓线
    Autodesk.Revit.DB.XYZ pnt1 = new Autodesk.Revit.DB.XYZ(0, 0, 0);
    Autodesk.Revit.DB.XYZ pnt2 = new Autodesk.Revit.DB.XYZ(1, 0, 0);
    Autodesk.Revit.DB.XYZ pnt3 = new Autodesk.Revit.DB.XYZ(1, 1, 0);
    Autodesk.Revit.DB.XYZ pnt4 = new Autodesk.Revit.DB.XYZ(0, 1, 0);
    Autodesk.Revit.DB.XYZ pnt5 = new Autodesk.Revit.DB.XYZ(0, 0, 1);
    CurveArrArray arrarr1 = new CurveArrArray();
    CurveArray arr1 = new CurveArray();
    arr1.Append(Line.CreateBound(pnt1, pnt2));
    arr1.Append(Line.CreateBound(pnt2, pnt3));
    arr1.Append(Line.CreateBound(pnt3, pnt4));
    arr1.Append(Line.CreateBound(pnt4, pnt1));
    arrarr1.Append(arr1);

    //创建顶部的轮廓线
    Autodesk.Revit.DB.XYZ pnt6 = new Autodesk.Revit.DB.XYZ(0.5, 0, 0);
    Autodesk.Revit.DB.XYZ pnt7 = new Autodesk.Revit.DB.XYZ(1, 0.5, 0);
    Autodesk.Revit.DB.XYZ pnt8 = new Autodesk.Revit.DB.XYZ(0.5, 1, 0);
    Autodesk.Revit.DB.XYZ pnt9 = new Autodesk.Revit.DB.XYZ(0, 0.5, 0);
    CurveArrArray arrarr2 = new CurveArrArray();
    CurveArray arr2 = new CurveArray();
    arr2.Append(Line.CreateBound(pnt6, pnt7));
    arr2.Append(Line.CreateBound(pnt7, pnt8));
    arr2.Append(Line.CreateBound(pnt8, pnt9));
    arr2.Append(Line.CreateBound(pnt9, pnt6));
    arrarr2.Append(arr2);

    //利用底部和顶部的轮廓线生成底部和顶部的轮廓
    SweepProfile bottomProfile = appCreator.NewCurveLoopsProfile(arrarr1);
    SweepProfile topProfile = appCreator.NewCurveLoopsProfile(arrarr2);

    //创建放样融合的拉伸路径
    Autodesk.Revit.DB.XYZ pnt10 = new Autodesk.Revit.DB.XYZ(5, 0, 0);
    Autodesk.Revit.DB.XYZ pnt11 = new Autodesk.Revit.DB.XYZ(0, 20, 0);
    Curve curve = Line.CreateBound(pnt10, pnt11);
```

```
//创建路径平面,CreateSketchPlane 参见 Extrusion 中的例子
    Autodesk.Revit.DB.XYZ normal =  Autodesk.Revit.DB.XYZ.BasisZ;
    SketchPlane sketchPlane =  CreateSketchPlane(normal, Autodesk.Revit.DB.XYZ.Zero);

    //利用底部和顶部的轮廓和拉伸路径创建放样融合
    SweptBlend newSweptBlend1 =  familyCreator.NewSweptBlend(true, curve, sketchPlane,
    bottomProfile, topProfile);
}
```

7.4.6　图元连接

上述五种图元创建方式是 Revit® 中支持的基本图元的创建方式。在构造一些复杂形体上,仅仅上述的创建方式是不够的,用户往往需要把几何图元连接起来构成复杂的形体。API 中有两个方法实现了连接几何图元和取消连接的功能。

Autodesk. Revit. DB. Document 类:

- GeomCombination CombineElements(CombinableElementArray members)
- void SeparateElements(CombinableElementArray members)

很显然,第一个函数是用来实现连接几何图元,而第二个函数是用来取消连接的。输入参数 CombinableElementArray 类是包涵多个 CombinableElement 实例的集合类。而 CombinableElement 类定义了可以被连接的图元的基类,也是上述五种图元的基类。

RevitSDK 中的 AutoJoin 示例代码是一个很好的学习材料,详见:Revit SDK 2015\Software Development Kit\Samples\FamilyCreation\AutoJoin\CS 文件夹。

7.4.7　小结

本节介绍了如何在族文档中使用 API 创建几何图元。通过把几何图元的属性关联到族参数是实现族文档中参数化建模的重要内容,也是 Revit® 参数化建模中的一个基础。

并不是所有的族文档都可以使用本节的 API 来创建几何图元的,如一些用来编辑轮廓线或者符号的族文档。一般来说,只有允许建模的族文档才能使用这些 API,用户可以通过文档的族类别来识别它们,如族类别是门或者家具的。另一个可能简单的判断方式是:该文档是否可以创建 3D 视图。

7.5　几何图元的可见性管理

在族文档中创建的图元(拉伸、融合等)会根据当前视图发生变化。在平面视图中,可能希望查看图元的二维表示。而在三维视图或立面视图中,则可能希望查看图元的三维表示的全部细节。可以灵活地显示详细程度不同的几何图形。在 UI 上,提供了对话框(图 7-12)来进行设置。

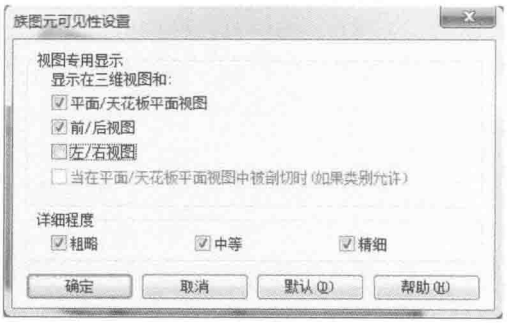


图 7-12　族图元可见性

在 API 中提供了 FamilyElementVisibility 类来完成相同的功能(设置可见性和详细程度),用户可以通过图元(Extrusion 类等)的 GetVisibility() 方法来获取该类,并通过

SetVisibility(FamilyElementVisibility visibility)方法来设置对应的值。

代码片段 7-19 示例了如何读取和设置图元的可见性。

代码片段 7-19:管理图元可见性

```
public void AccessFamilyElementVisibility(Extrusion extrusion)
{
    //得到管理拉伸体的可见性的实例,并读取详细程度的设置
    FamilyElementVisibility visibility = extrusion.GetVisibility();
    FamilyElementVisibilityType visibilityType = visibility.VisibilityType;
    bool shownInCoarse = visibility.IsShownInCoarse;
    bool shownInMedium = visibility.IsShownInMedium;
    bool shownInFine = visibility.IsShownInFine;

    //设置为在各种详细程度中都显示拉伸体
    visibility.IsShownInCoarse = true;
    visibility.IsShownInMedium = true;
visibility.IsShownInFine = true;
//注意:必须把可见性的修改设置回拉伸体
    extrusion.SetVisibility(visibility);
}
```

7.6 编辑族和载入族

编辑族允许打开当前文档中已经加载的族文档来编辑,而载入族允许把新创建或者修改过的族文档载入到需要的文档中,这个文档可能是个工程文档或者族文档。

例如,在工程文档中的"编辑族"和在族文档中的"载入到项目中"是 Revit® 中族文档和工程文档交互的重要方式,API 提供了函数实现了类似的功能。

7.6.1 编辑族

Autodesk. Revit. DB. Document 类:Document EditFamily(Family loadedFamily)。

该函数会打开输入的族的文档,并返回族文档的实例。然后 API 用户就可以自由的调用 API 来实现族文档的各类修改。

注意:该函数打开的文档是需要使用 API 来进行关闭或者载入,这个打开的文档不会在 UI 界面上显示。

7.6.2 载入族

Autodesk. Revit. DB. Document:

- Family LoadFamily(Document targetDocument);
- Family LoadFamily(Document targetDocument, IFamilyLoadOptions familyLoadOptions);

这里有两个函数实现载入族的功能。该函数会把族文档载入到目标文档,并返回目标文档中载入族的实例。这两个函数应用于不同的场景中,第一个函数只需要一个目标文档作为参数,但是该函数只适用于把族文件载入到一个从未加载过该族文件的目标文档中,否则该函数会直接失败掉,因为该函数不能处理已加载的族和新加载的族之间的任何冲突。而第二个函数多了一个 IFamilyLoadOptions 的接口作为参数,该接口给了用户处理加载过

程中的冲突的方式。另外，API 中提供了一个方法来获取和 UI 界面(图 7-13)一致的 IFamilyLoadOptions 的实现。

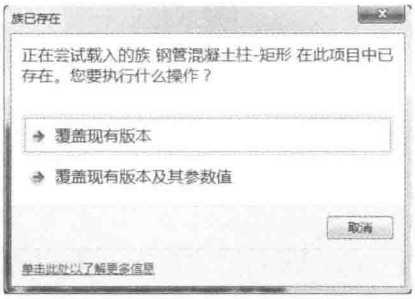


图 7-13　加载族选项

Autodesk. Revit. UI. UIDocument 类：IFamilyLoadOptions GetRevitUIFamilyLoadOptions()。

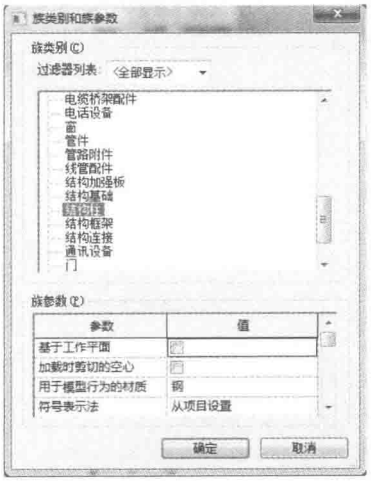


图 7-14　族类别和族参数

7.7　其他

7.7.1　族类别和族参数

如图 7-14 所示，一个族文档必然有一个族类别和一系列族参数，用户可以通过图 7-14 的方式来访问或者修改它们，那么在 API 中，如何取得该数据呢？可以通过 Document. OwnerFamily 属性来获取族文档的定义的族，然后通过 Family. FamilyCategory 属性来获取族类别信息，而族参数信息就存储在 Family. Parameters 属性中。

7.7.2　概念体量

在 Revit® 中，有一个较特殊的族文档——概念体量。它创建几何图元的方式和其他族文档完全不同，所以本节的内容不适用“概念体量”的族文档。API 有另一套函数来实现“概念体量”中的几何图元的创建和编辑。该内容在本书中不重点介绍，仅给出函数列表。

Autodesk. Revit. Creation. FamilyItemFactory 类：

- Form NewExtrusionForm(bool isSolid, ReferenceArray profile, XYZ direction);
- Form NewFormByCap(bool isSolid, ReferenceArray profile);
- Form NewFormByThickenSingleSurface(bool isSolid, Form singleSurfaceForm, XYZ thickenDir);
- Form NewLoftForm(bool isSolid, ReferenceArrayArray profiles);
- FormArray NewRevolveForms(bool isSolid, ReferenceArray profile, Reference axis, double startAngle, double endAngle);
- Form NewSweptBlendForm(bool isSolid, ReferenceArray path, ReferenceArrayArray profiles);
- Control NewControl(ControlShape controlShape, View view, XYZ origin);
- CurveByPoints NewCurveByPoints(ReferencePointArray points);

第 8 章　视图(Views)

视图是通过 Revit® 的模型创建的图像，它是对 Revit® 文档中的数据的图像化展示。视图可以是图形，如平面视图；也可以是文字，如明细表。每个 Revit® 文档可以有一个或多个视图。当前获得焦点的窗口被称为活动视图(ActiveView)。

本章将介绍视图是如何生成的，Revit® 中支持哪些类型的视图，以及各种视图的特性。

8.1　概述

本节将概括地讨论如下内容：

- 视图是如何生成的；
- 视图的类型；
- 元素在视图中的可见性；
- 创建和删除视图。

8.1.1　视图处理

图 8-1 展示了一个视图生成的过程。

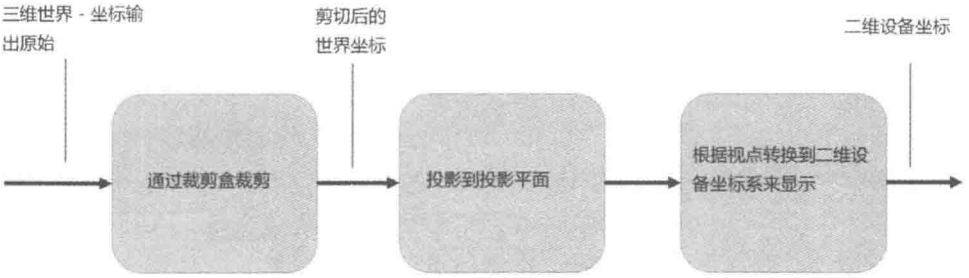


图 8-1　视图生成过程

每个视图的生成都是一个将三维物件投射到二维平面上的过程。有两种类型的投影：透视投影和正交投影。

在确定了投影类型后，必须设置所需三维模型的条件和渲染的场景。有关投影的更多介绍，请参考“8.2 三维视图(View3D)”。

世界坐标系包括以下两部分：视点的位置和观察平面(也就是投影平面的位置)。

Revit® 使用两种坐标系：世界或模型空间坐标系和观察坐标系。

世界坐标系是系统的绝对坐标系，在没有建立用户坐标系之前画面上所有点的坐标都是以该坐标系的原点来确定各自的位置的。

观察坐标系用来将模型呈现在观察者眼中。它的原点就是观察者的视点，通过 View. Origin 属性来获取。View. RightDirection，View. UpDirection 和 View. ViewDirection 分别表示 X，Y 和 Z 轴。

- View. RightDirection 的方向朝向屏幕的右方。
- View. UpDirection 的方向朝向屏幕的上方。
- View. ViewDirection 的方向从屏幕射向观察者。

观察坐标系遵从右手定则。更多的介绍请参考“8. 2 三维视图(View3D)”的透视投影图和正交投影图的内容。

有些模型或模型的一部分不会显示出来，比如在观察者后面的模型，或者在很远处无法显示清楚的模型，它们不会被投影的投影平面。通过裁剪可以达到这个目的。以下是裁剪的规则：

- 在裁剪盒外的元素不会显示在视图中。
- View. CropBox 属性提供了裁剪盒的几何信息。它返回一个观察坐标系的平行六面体 BoundingBoxXYZ。坐标系和裁剪盒会在“8. 2 三维视图(View3D)”介绍。

在裁剪后，模型会被投射到投影平面。以下是投影的规则：

- 投影的内容被映射到屏幕的可见区域来显示出来。
- 在映射过程中，投影的内容会被缩放来使它们正确的显示在屏幕上。
 - View. Scale 属性代表实际模型和视图模型的比例；
 - 视图的边界(裁剪)是裁剪盒在投影平面上的投影；
 - 裁剪区域的尺寸和位置可以通过 View. OutLine 属性获得。

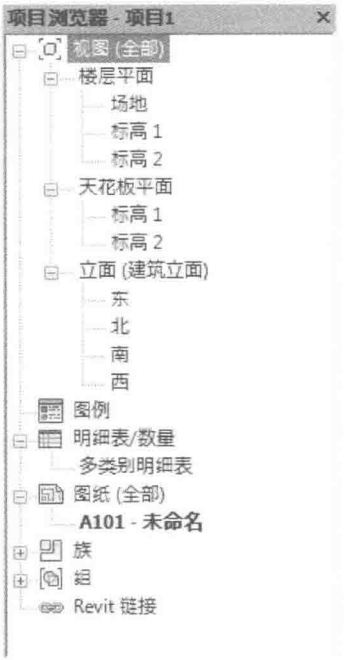


图 8-2　项目浏览器中的各种视图

8.1.2　视图类型

一个 Revit® 工程可以包含几种视图类型。图 8-2 展示了项目浏览器中的各种不同的视图。

在 API 中有两种方法对所有的视图进行分类。第一种是通过视图元素的 ViewType 属性，它返回一个枚举值来表示视图的类型。表 8-1 列出了所有可用的视图类型。

表 8-1　　可用视图类型表

视图类型	描述
AreaPlan	面积平面视图
CeilingPlan	天花板投影平面视图
ColumnSchedule	柱明细表视图
CostReport	成本报告视图
Detail	详图视图
DraftingView	绘图视图
DrawingSheet	图纸
Elevation	立面视图

续表

视图类型	描述
EngineeringPlan	结构平面视图或建造视图
FloorPlan	楼层平面视图
Internal	内部视图
Legend	图例视图
LoadsReport	负荷报告视图
PanelSchedule	配电盘明细表
PresureLossReport	压力损失报告视图
ProjectBrowser	项目浏览器视图
Rendering	渲染视图
Report	报告视图
Schedule	明细表视图
Section	截断面视图
SystemBrowser	系统浏览器视图
ThreeD	三维视图
Undefined	未定义视图
Walkthrough	漫游视图

第二种是通过视图类的类型来分类。图 8-3 展示了 Revit® API 中视图类的架构。

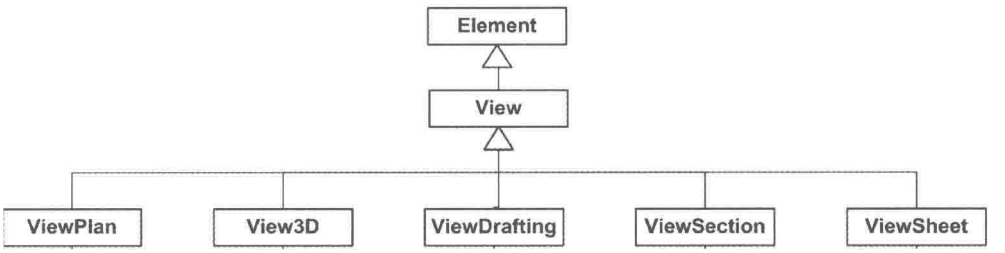


图 8-3　视图类架构图

表 8-2 列出了视图的类型、视图对应的类和在项目浏览器中对应的视图。

表 8-2　　项目浏览器中的视图对应的视图类型和视图类列表

项目浏览器中的视图	视图类型	视图类
面积平面	ViewType. AreaPlan	ViewPlan
天花板平面	ViewType. CeilingPlan	ViewPlan
图形柱明细表	ViewType. ColumnSchedule	View
报告	ViewType. CostReport	View
详图视图	ViewType. Detail	ViewSection
绘图视图	ViewType. DraftingView	ViewDrafting
图纸	ViewType. DrawingSheet	ViewSheet

续表

项目浏览器中的视图	视图类型	视图类
立面	ViewType. Elevation	View
结构平面	ViewType. EngineeringPlan	ViewPlan
楼层平面	ViewType. FloorPlan	ViewPlan
(内部类型)	ViewType. Internal	View
图例	ViewType. Legend	View
报告 (Revit® MEP)	ViewType. LoadsReport	View
报告 (Revit® MEP)	ViewType. PanelSchedule	View
报告 (Revit® MEP)	ViewType. PresureLossReport	View
项目浏览器	ViewType. ProjectBrowser	View
渲染	ViewType. Rendering	ViewDrafting
报告	ViewType. Report	View
明细表/数量	ViewType. Schedule	ViewSchedule
剖面	ViewType. Section	ViewSection
系统浏览器	ViewType. SystemBrowser	View
三维视图	ViewType. ThreeD	View3D
(未定义)	ViewType. Undefined	—
漫游	ViewType. Walkthrough	View3D

代码片段 8-1 展示了如何获得视图类型和视图的类类型。

代码片段 8-1:获取视图类型

```
//Autodesk.Revit.DB.View view =  GetView();
//两种判断视图类型的方法:
//第一种:
ViewType viewType =  view.ViewType;
switch (viewType)
{
case Autodesk.Revit.DB.ViewType.ThreeD:
    //视图类型是三维视图
    break;
//其他类型
}

//第二种:
if (view is View3D)
{
// view 的类类型是三维视图
}
```

8.1.3　视图中元素的可见性

视图中记录了所有可见的元素。通过 FilteredElementCollector 类可以获得所有在一个视图中的可见的有图形的元素,见代码片段 8-2。当然,有些元素可能被设置成隐藏或被其他元素遮挡。要想显示它们,可以不隐藏它们或旋转视图等。

可以通过类别(Category)来设置视图中某类元素的显示或隐藏。

- View. GetVisibility(Category category)查询一个类别在这个视图里是否显示。
- View. SetVisibility(Category category, bool visible)设置一个类别在这个视图里是否显示,所有属于这个类别的元素会同时隐藏或显示。

代码片段 8-2:获取视图中可见的元素

```
//找到视图中所有可见的元素
FilteredElementCollector elemCollector =  new FilteredElementCollector(document, viewId);
foreach (Element elem in elemCollector)
{
    //操作元素 elem
}
```

8.1.4 创建和删除视图

在 Revit® API 中,每个视图类都有相应的函数来创建视图对象,见表 8-3。

表 8-3　　视图创建的方法及参数表

类	方法	参数
View3D	View3D. CreateIsometric:创建正交三维视图	• Document document: Revit 文档对象 • ElementId viewFamilyTypeId: ViewFamilyType 对象的 Id,必须是三维(ThreeDimensional)视图类型
View3D	View3D. CreatePerspective:创建透视三维视图	• Document document: Revit 文档对象 • ElementId viewFamilyTypeId:ViewFamilyType 对象的 Id,必须是三维(ThreeDimensional)视图类型
ViewDrafting	Creation. Document. NewViewDrafting	—
ViewPlan	ViewPlan. Create:创建平面视图	• Document document: Revit 文档对象 • ElementId viewFamilyTypeId:ViewFamilyType 对象的 Id,必须是楼板平面(FloorPlan),天花板平面(CeilingPlan),面积平面(AreaPlan),或者结构平面(StructuralPlan)的视图类型 • ElementId levelId: 标高的 Id
	ViewPlan. CreateAreaPlan:创建面积平面视图	• Document document: Revit 文档对象 • ElementId areaSchemeId: 面积模式(AreaSchema)的 Id • ElementId levelId:标高的 Id
ViewSchedule	ViewSchedule. CreateViewList: 创建视图列表	• Document document:Revit 文档对象
	ViewSchedule. CreateKeynoteLegend:创建注释记号图例	• Document document:Revit 文档对象
	ViewSchedule. CreateKeySchedule: 创建关键字明细表	• Document document: Revit 文档对象 • ElementId categoryId:明细表所对应的类别的元素 Id 值
	ViewSchedule. CreateMaterialTakeoff:创建材质提取明细表	• Document document:Revit 文档对象 • ElementId categoryId:明细表所对应的类别的元素 Id 值

续表

类	方法	参数
ViewSchedule	ViewSchedule. CreateNoteBlock：创建注释块	• Document document：Revit 文档对象 • ElementIdfamilyId：明细表所对应的族类别的元素 Id 值
	ViewSchedule. CreateRevision Schedule：创建修订明细表	• Document document：Revit 文档对象
	ViewSchedule. CreateSchedule：创建明细表	• Document document：Revit 文档对象
	ViewSchedule. CreateSchedule：创建明细表	• Document document：Revit 文档对象 • ElementId categoryId：明细表所对应的类别的元素 Id 值
	ViewSchedule. CreateSheetList：创建图纸列表	• Document document：Revit 文档对象
ViewSection	View. CreateCallout：创建详图索引视图	• Document document：Revit 文档对象 • ElementId parentViewId：详图索引视图所在的视图 • ElementId viewFamilyTypeId：详图索引视图的族类型 • XYZ point1，XYZ point2：详图索引视图在所在视图的位置
	ViewSection. CreateDetail：创建详图视图	• Document document：Revit 文档对象 • ElementId viewFamilyTypeId：详图索引视图的族类型 • BoundingBoxXYZ sectionBox：详图视图的范围
ViewSheet	ViewSheet. Create：创建图纸视图	• Document document：Revit 文档对象 • ElementId titleBlockTypeId：标题栏类型
PanelScheduleView	PanelScheduleView. CreateInstanceView：创建配电盘明细表	• Document document：Revit 文档对象 • ElementId panelId：配电盘元素 Id
	PanelScheduleView. CreateInstanceView：创建配电盘明细表	• Document document：Revit 文档对象 • ElementId templateId：视图模板 Id • ElementId panelId：配电盘元素 Id

如果创建成功，这些方法会返回相应对象的引用，如果失败，会抛出异常或返回空值。这些方法的使用会在下面的章节详细介绍。

要删除一个视图可以使用 Document. Delete 方法，传入一个视图的 ID。也可以通过删除该视图所依赖的对象来间接删除该视图，例如，删除标高，相应的平面视图也会被删除，或者删除相机，相应的三维视图也会被删除。

8.2　三维视图(View3D)

View3D 类代表 Revit® 中的三维视图。Revit® 中有两种三维视图：透视三维视图和正交三维视图。这两种视图的区别在于投影射线的关系。属性 View3D. IsPerspective 表示一个三维视图是透视视图或正交视图。

可以通过下面的步骤在 Revit® 中添加一个相机来创建一个三维视图：

- 点击“视图”选项卡→“创建”面板→“三维视图”下拉框→“相机”。
- 勾选“透视”复选框来添加一个透视三维视图或者不勾选来添加一个正交三维视图(图 8-4)。
- 点击绘图区域来添加一个相机。
- 选一个点来指定观察方向。



图 8-4 透视图勾选框

8.2.1 透视三维视图(Perspective View3D)

可以通过 View3D.CreatePerspective 来创建一个透视三维视图。

图 8-5 展示了一个透视三维视图是如何创建的。

模型上的每个点通过直线投影线投影到投影平面来形成投影的内容。为了方便将世界坐标系转换到视图平面上,观察坐标系是基于观察者建立的。原点,也就是 View.Origin 属性,就是观察者的位置。观察者的世界坐标系可以通过 View3D.EyePosition 属性得到。因此,在三维视图中,View.Origin 总是和 View3D.EyePosition 相同。

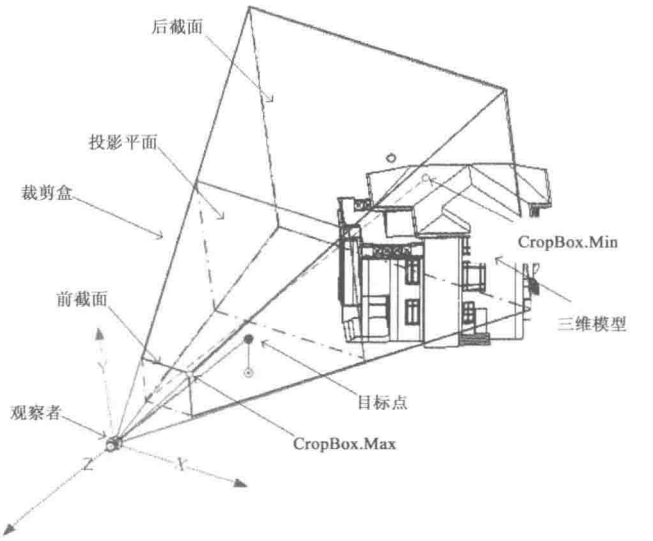


图 8-5 透视投影

正如上述,观察坐标系由下面这些属性决定:

- *X* 轴由 View.RightDirection 决定;
- *Y* 轴由 View.UpDirection 决定;
- *Z* 轴由 View.ViewDirection 决定。

View.ViewDirection 在三维空间中是从目标点射向观察者,在屏幕空间中是从平面射向观察者。

透视视图的裁剪盒是以观察者位置为顶点的观察棱锥的一部分,通过两个平行的平面裁剪而成。裁剪盒划定了模型被裁剪掉的部分和投影到观察平面的部分。

裁剪盒通过 View.CropBox 属性获得,它返回一个 BoundingBoxXYZ 对象。

图 8-5 中表示了 CropBox.Min 和 CropBox.Max 两个点,CropBox.Min 这个点是由裁剪盒的 CropBox.Max 通过从前截面向后截面投影而获得的。

裁剪盒的坐标使用的是视图坐标系。用 Transform.OfPoint 可以将 CropBox.Min 和 CropBox.Max 转换到世界坐标系。具体内容参见“第 6 章几何(Geometry)”。

投影平面以及前、后两个截面都与观察方法垂直。CropBox.Max 和 CropBox.Min 的

连线和观察方法平面，这样裁剪盒里的几何图形就能被计算出来了。

图 8-6 是裁剪后的投影平面在屏幕上的显示。裁剪区域就是裁剪盒和投影平面相交的矩形区域。

- 裁剪区域的几何信息可以通过 View. CropRegion 属性获得，它返回一个 BoundingBoxUV 对象。
- View. Outline. Max 属性返回右上角的点。
- View. Outline. Min 属性返回左下角的点。

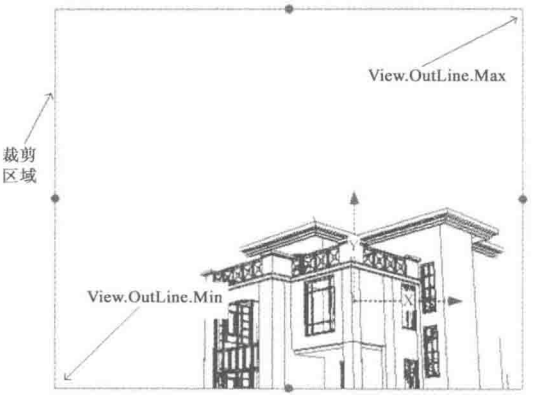


图 8-6　透视三维视图

和裁剪盒一样，裁剪区域的坐标使用的是视图坐标系。下面的等式表达的这些属性之间的关系：

View. CropBox. Max. X(Y) / View. OutLine. Max. X(Y) == View. CropBox. Min. X(Y) / View. OutLine. Min. X(Y)

因为一个物体的各个部分在透视视图中大小遵循近大远小的规则，所以视图的比例值对透视视图来说是无意义的。透视视图的 Scale 属性总是返回 0。

8.2.2　正交三维视图(Orthographic View3D)

View3D 提供了函数 CreateIsometric 来创建正交三维视图(图 8-7)。

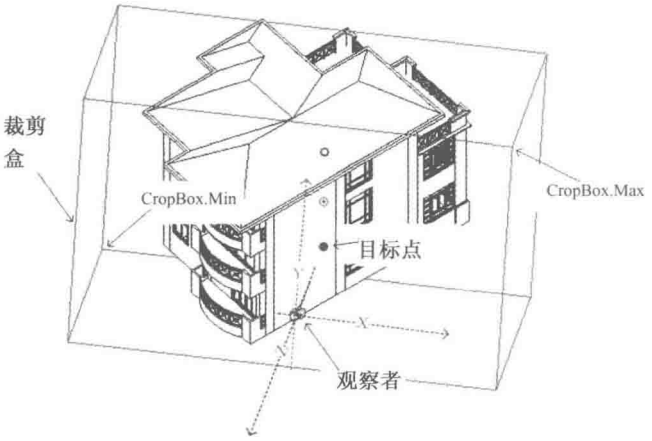


图 8-7　正交投影

正交三维视图是通过平行投影将模型投影到平面上。视图坐标系和投影三维视图相像，但裁剪盒的各个面要和投影线平行，或者和投影线垂直。View. CropBox 属性指向两个基于视图坐标系的对角线坐标。

模型被投影到视图平面，然后通过缩放显示在屏幕上(图 8-8)。View. Scale 属性代表实际模型尺寸和视图上显示的尺寸直接的比例。它们的关系如下：

View. CropBox. Max. X(Y) / View. OutLine. Max. X(Y) == View. CropBox. Min. X(Y) / View. OutLine. Min. X(Y) == View. Scale

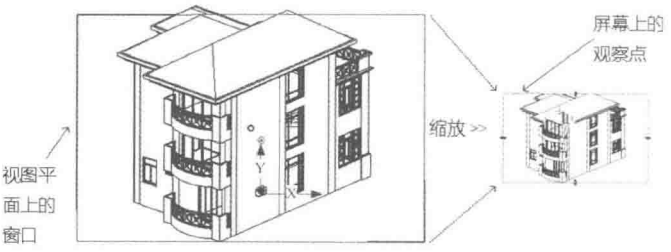


图 8-8　缩放视图平面上的窗口到屏幕观察点

可以通过 View3D. CreateIsometric(Document document, ElementId viewFamilyTypeId),传入文档对象和视图族类型的 ID 来创建一个正交三维视图。Revit® 自动定义下面的内容:

- 观察者的位置
- 如何创建视图坐标系
- 如何创建裁剪盒来剪切模型

一旦视图创建好,就可以通过 View3D. SetOrientation 来改变视图坐标系,通过改变裁剪盒的大小来看模型的不同部分。

代码片段 8-3 展示了如何创建一个正交三维视图。

代码片段 8-3:创建一个正交三维视图

```
private void CreateView3D(Autodesk.Revit® .Document doc, ElementId viewTypeId)
{
  try
  {
     // Create a new View3D
     XYZ direction =  new XYZ(1, 1, 1);
     View3D view3D =  View3D.CreateIsometric(doc, viewTypeId);
     if (null = =  view3D)
      return;

     // The created View3D isn't perspective.
     Debug.Assert(false = =  view3D.IsPerspective);
  }
  catch (Exception e)
  {
     Debug.WriteLine(e.ToString());
  }
}
```

8.2.3 三维视图剖面框(Section Box)

每个视图都有一个裁剪盒。它的作用就是决定模型在视图中显示的哪部分。对三维视图来说,有另一个概念:剖面框(图 8-9)。

- 剖面框决定了模型的哪些部分显示在三维视图中。
- 剖面框用来裁剪三维模型的可见部分。
- 在剖面框外面的部分即使在裁剪盒中也不可见。
- 和裁剪盒不同的是剖面框可以随模型一起移动和旋转。

剖面框对大模型来说非常有用。例如,如果想要渲染一幢非常大的建筑,可以使用剖面框。它可以减少光能传递和光线跟踪的计算。如果要显示剖面框,可以在三维视图

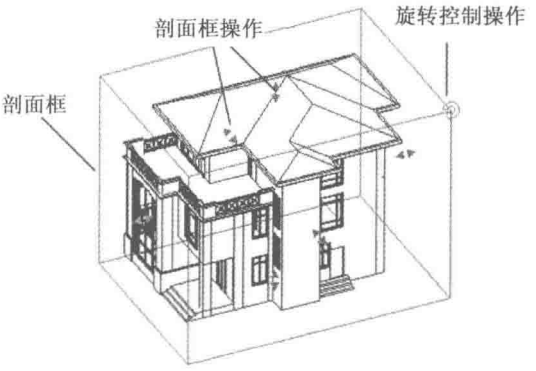


图 8-9 剖面框

的属性框中选中剖面框选项。代码片段 8-4 展示了如何使用 API 来设置。

代码片段 8-4:显示剖面框

```
private void ShowHideSectionBox(View3D view3D)
{
  view3D.IsSectionBoxActive =  true;
}
```

8.2.4　三维视图的旋转

在三维视图中通过三维导航工具可以任意旋转该视图。在 API 中,可以通过 View3D.SetOrientation 来实现同样的功能。

- View3D. SetOrientation(ViewOrientation3D newViewOrientation3D)

其中,newViewOrientation3D 参数是需要提供的新的三维坐标,它的类型是 ViewOrientation3D。这个类型的对象包含以下信息:

- EyePosition:观察者所在位置
- ForwardDirection:观察者需要观察的方向
- UpDirection:观察者向上的方向

另外,相对应地,View3D. GetOrientation 函数可以得到当前观察者的坐标。

注意:这只是临时改变三维视图中的观察坐标,改变后的视图坐标不会被保存的 Revit® 文件中。

8.3　平面视图(ViewPlan)

平面视图都是基于标高来创建的,有两种类型:楼层平面视图和天花板平面视图。

- 通常,楼层平面视图是在新建一个工程时打开的默认视图。
- 大多数工程包括至少一个楼层平面视图和一个天花板平面视图。
- 平面视图通常都通过标高来创建的。

通过 API 创建标高不会自动创建平面视图。ViewPlan 类提供了 Create 方法来创建一个平面视图:

- ViewPlan. Create(Document document, ElementId viewFamilyTypeId, ElementId levelId)

其中,viewFamilyTypeId 参数是视图类型(ViewFamilyTyp)的 Id。该对象的视图类型(ViewType)必须是楼层平面(FloorPlan),天花板平面(CeilingPlan),面积平面(AreaPlan),或者结构平面(StructuralPlan)中的一种。参数 levelId 代表在这个工程中的和欲创建的平面视图相关联的标高。代码片段 8-5 通过一个标高来创建相应的楼层平面和天花板平面。

代码片段 8-5:创建楼层平面和天花板平面

```
private void CreatePlanView(Autodesk.Revit.Document doc, ElementId viewTypeId)
{
  try
  {
    using(Transaction tr =  new Transaction(doc))
```

```
    {
     tr.Start("创建平面视图");
     double elevation = 10.0;
     Level level1 = doc.Create.NewLevel(elevation);
     ViewPlan floorView = ViewPlan.Create(doc, viewTypeId, level1.Id);
     tr.Commit();
    }
  }
  catch (Exception exp)
  {
      MessageBox.Show(exp.ToString());
  }
}
```

8.4 绘图视图(ViewDrafting)

绘图视图是与模型没有直接关联的详图。可以用 Autodesk. Revit. Creation. NewViewDrafting 来创建一个绘图视图。

① 在绘图视图中,用户可以创建不同的详细程度(粗略、中等、精细)。

② 可使用二维详图工具:

- 详图线
- 详图区域
- 详图构件
- 隔热层
- 参照平面
- 尺寸标注
- 符号
- 文字

③ 绘图视图不显示任何模型元素。

8.5 剖面视图(ViewSection)

剖面视图可以剪切模型来显示类别结构。可以用 ViewSection. CreateSection 方法来创建剖面视图。

- ViewSection. CreateSection(Document document, ElementId viewFamilyTypeId, BoundingBoxXYZ sectionBox)

其中,sectionBox 参数是剖面视图的裁剪盒,它决定了该视图的方向和范围。通常可以使用另一个视图的裁剪盒作为参数,也可以自定义一个 BoundingBoxXYZ 实例来确定视图

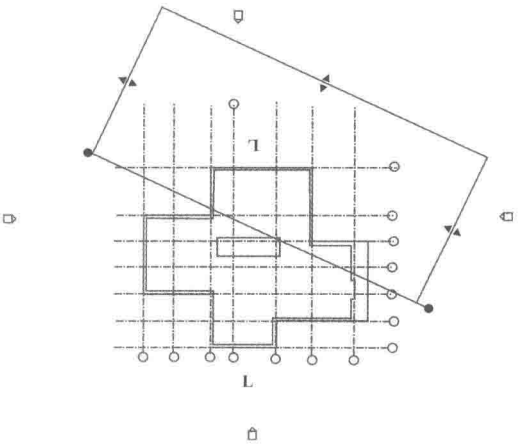


图 8-10 平面视图中的剖面表示

的方向和范围。

图 8-10 显示的是平面视图中的剖面表示，图 8-11 显示了一个剖面视图。

如图 8-12 所示，O-XYZ 是模型坐标系统，裁剪盒的 Max 和 Min 坐标是基于该坐标系统的，当它被用到创建剖面视图后，剖面视图会创建一个新的坐标系统 O′-X′Y′Z′作为观察坐标系。

- O′是剖面视图的原点，它的坐标通过 View. Origin 属性获得。
- 剖面视图的裁剪盒通过传入的参数计算得到。

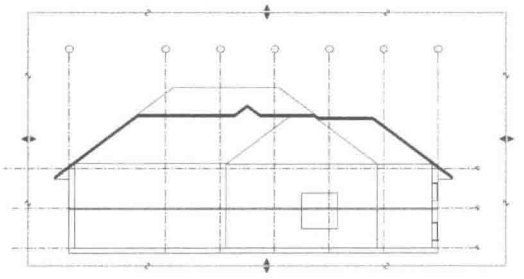


图 8-11　剖面视图

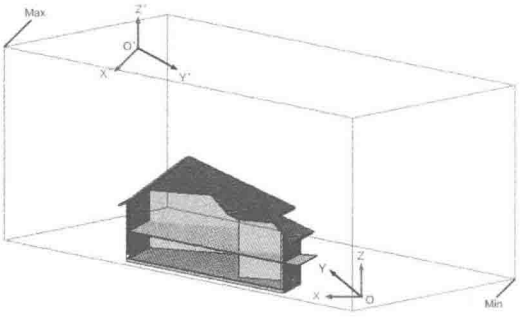


图 8-12　剖面视图裁剪盒

8.6　参照详图索引视图和详图视图

详图视图(图 8-13)是一个在其他视图中显示为详图索引或剖面的模型视图。这种类型的视图通常表示的模型的详图比例比在父视图中更精细。从剖面视图、平面视图或立面视图创建详图索引时，可以将视图链接到建筑模型的某个特定区域。

可以用 ViewSection. CreateCallout 和 ViewSection. CreateDetail 来创建参照详图索引视图和详图视图。

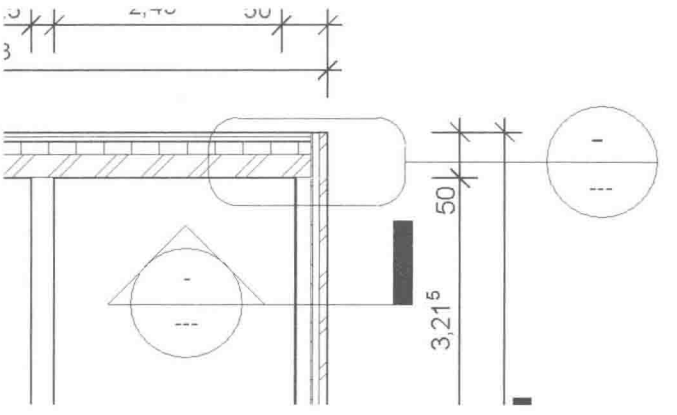


图 8-13　详图索引视图和详图视图

详图索引以较大比例显示另一视图的一部分。在施工图文档集中，使用详图索引可以持续增加的详细程度提供标记视图的有序变化。

• View. CreateCallout (Document document, ElementId parentViewId, ElementId viewFamilyTypeId, XYZ point1, XYZ point2)

其中,point1 和 point2 参数是详图符号在父视图中的范围。

详图视图是一个在其他视图中显示为详图索引或剖面的模型视图。这种类型的视图通常表示的模型的详图比例比在父视图中更精细。该视图用于将详细信息添加到模型的特定部分。

• ViewSection. CreateDetail (Document document, ElementId viewFamilyTypeId, BoundingBoxXYZ sectionBox)

所创建视图的观察方向由 sectionBox. Transform. BasisZ 决定,向上方向由 sectionBox. Transform. BasisY 决定,向右方向通过左手定则计算出来。裁剪区域通过 BoundingBoxXYZ. Min 和 BoundingBoxXYZ. Max 投影到视图的裁剪平面上来获得。裁剪距离由 BoundingBoxXYZ. Min 和 BoundingBoxXYZ. Max 在 Z 坐标相减得到。新创建的 ViewSection 实例会自动分配一个唯一的视图名。

8.7 图纸视图(ViewSheet)

图纸是施工图文档集的一个独立的页面。在 Revit® 中,为施工图文档集中的每个图纸创建一个图纸视图。然后在每个图纸视图上放置多个图形或明细表。

当使用 ViewSheet. Create 方法来创建一个图纸视图时,需要传入一个标题栏族类型。可以通过 Autodesk. Revit. Document. TitleBlocks 属性来获取当前文档中所有标题栏对象。

新创建的图纸里是没有视图的。可以通过 Viewport. Create 来放置某个视图到图纸中。

• Viewport. Create (Document document, ElementId viewSheetId, ElementId viewId, XYZ point)

其中,viewSheetId 是用来放置视图的图纸,viewId 是需要被放置的视图,point 是放置的位置。

每个图纸都有一个唯一的图纸号。这个图纸号显示在项目浏览器里的图纸名称前。可以很方便地使用图纸号在各个图纸中交叉引用。可以通过 ViewSheet. SheetNumber 来获取或修改图纸号。

代码片段 8-6 展示了如何创建和打印一个图纸视图。先找一个可用的标题栏来创建一个图纸视图。然后,将当前活动视图加入到图纸视图的中心位置。最后,将图纸视图打印出来。

代码片段 8-6:创建和打印一个图纸视图

```
private void CreateSheetView(Autodesk.Revit.Document doc)
{
  // Get an available title block from document
  FamilySymbolSet fsSet =  doc.TitleBlocks;
  if (fsSet.Size = =  0)
  {
```

```
        MessageBox.Show("No title blocks");
        return;
    }

    FamilySymbol fs = null;
    foreach (FamilySymbol f in fsSet)
    {
        if (null != f)
        {
          fs = f;
          break;
        }
    }

    try
    {
        // Create a sheet view
        ViewSheet viewSheet = ViewSheet.Create(doc, fs);
        if (null == viewSheet)
          return;

        // Add current view onto the center of the sheet
        UV location = new UV(
          (viewSheet.Outline.Max.U-viewSheet.Outline.Min.U) / 2,
          (viewSheet.Outline.Max.V-viewSheet.Outline.Min.V) / 2);

        XYZ point = new XYZ(UV.U, UV.V, 0);
        Viewport.Create(doc, viewSheep.Id, doc.ActiveView.Id, point);

        // Print the sheet out
        if (viewSheet.CanBePrinted)
        {
        if (MessageBox.Show("Print the sheet?", "Revit®", MessageBoxButtons.YesNo) == =
        DialogResult.Yes)
          viewSheet.Print();
        }
    }
    catch (Exception e)
    {
        MessageBox.Show(e.ToString());
    }
}
```

注意:一个图纸视图不能被加入另一个图纸视图,也不能将一个视图重复加入多个图纸视图。

8.8　明细表(ViewSchedule)

明细表(图 8-14)以表格形式显示信息,这些信息是从项目中的图元属性中提取的。明细表可以列出要编制明细表的图元类型的每个实例,或根据明细表的成组标准将多个实例

压缩到一行中。

房间明细表						
				涂层		
编号	面积	体积	占用	楼板涂层	墙涂层	天花板涂层
5	115.37 SF	1673 CF	Shared	Ceramic Tile	White Painted	Acoustic Tile 2'x2'
27	1988.39 SF	28833 CF	Shared	Ceramic Tile	White Painted	Acoustic Tile 2'x2'
循环：2	2103.76 SF					
10	436.32 SF	6327 CF	Office	Ceramic Tile	Light Blue Painted	Acoustic Tile 2'x2'
13	313.14 SF	4541 CF	Office	Ceramic Tile	Light Blue Painted	Acoustic Tile 2'x2'
14	358.36 SF	5196 CF	Office	Ceramic Tile	Light Blue Painted	Acoustic Tile 2'x2'
15	350.66 SF	5085 CF	Office	Ceramic Tile	Light Blue Painted	Acoustic Tile 2'x2'
17	235.44 SF	3414 CF	Office	Ceramic Tile	Light Blue Painted	Acoustic Tile 2'x2'
18	235.44 SF	3414 CF	Office	Ceramic Tile	Light Blue Painted	Acoustic Tile 2'x2'
21	265.59 SF	3851 CF	Office	Ceramic Tile	Light Green Painted	Acoustic Tile 2'x2'
22	235.44 SF	3414 CF	Office	Ceramic Tile	Light Green Painted	Acoustic Tile 2'x2'
25	268.48 SF	3893 CF	Office	Ceramic Tile	Light Green Painted	Acoustic Tile 2'x2'
26	262.69 SF	3809 CF	Office	Ceramic Tile	Light Green Painted	Acoustic Tile 2'x2'
办公室：10	2961.54 SF					

图 8-14　明细表示例

通过 API 可以创建下面这些类型的明细表：

- 明细表(或数量)
- 关键字明细表
- 材质提取
- 注释明细表(或注释块)
- 修订明细表(请参见图纸上的修订明细表)
- 视图列表(请参见视图列表)
- 图纸列表(请参见图纸列表)
- 配电盘明细表(MEP)

普通明细表可以通过 ViewSchedule. CreateSchedule 函数来创建。

- ViewSchedule. CreateSchedule (Document document, ElementId categoryId)

其中，categoryId 用来指定哪个类别的元素被统计在明细表中，或者传入 ElementId. InvalidElementId 来指定多个类别。

对于面积类别的明细表，还需要指定面积结构(Area Sheme)元素的 ID。

- ViewSchedule. CreateSchedule (Document document, ElementId categoryId, ElementId areaSchemeId)

明细表可以包含多个具有相同特征的项目。可以使用关键字明细表定义关键字。创建关键字时，关键字会作为图元的实例属性列出。当应用关键字的值时，关键字的属性将应用到图元中，可以自动填充信息。关键字明细表可以通过 ViewSchedule. CreateKeySchedule 函数来创建。

- ViewSchedule. CreateKeySchedule (Document document, ElementId categoryId)

其中，document 指定创建的明细表所在的文档，dategoryId 指定元素的类别。

材质提取明细表列出所有 Revit® 族的子构件或材质。材质提取明细表具有其他明细表视图的所有功能和特征，但是可用其更详细地显示构件的部件的信息。Revit® 中的构件的任何材质都可以显示在明细表中。材质提取明细表可以通过 ViewSchedule.

CreateMaterialTakeoff 函数来创建。

• ViewSchedule. CreateMaterialTakeoff (Document document, ElementId categoryId)

其中,document 指定创建的明细表所在的文档,dategoryId 指定元素的类别。

注释明细表或注释块对于列出项目中应用到图元的注释是很有用的。例如,如果希望将一个注释附着到几面墙上,该注释可能对每面墙都进行了建筑说明。要创建一个注释明细表可以使用 ViewSchedule. CreateNoteBlock 函数。

• ViewSchedule. CreateNoteBlock (Document document, ElementId familyId)

其中,document 指定创建的明细表所在的文档,familyId 指定哪个族的图元包含在该明细表中。

大多数图纸的标题栏都包含修订明细表。将视图放置在图纸上并且该视图包含修订云线时,修订明细表将自动显示有关这些修订的信息。如果需要,可以指定将其他修订(不是由视图中的修订云线表示)显示在修订明细表中。

要在图纸上显示修订明细表,可使用包含修订明细表的标题栏。当设计修订明细表时,可以指定其格式、页面上的方向、显示的信息以及排序顺序。可以通过修改标题栏族来控制修订明细表的这些属性。

要创建一个修订明细表,可以使用 ViewSchedule. CreateRevisionSchedule 函数。

• ViewSchedule. CreateRevisionSchedule(Document document)

其中,document 是标题栏的族文档。

视图列表是项目中视图的明细表。在视图列表中,可按类型、标高、图纸或其他参数对视图进行排序和分组。如果需要,可在图纸中包含视图列表。

视图列表可帮助您执行下列操作:

• 管理项目中的视图。

• 跟踪视图的状态。

• 确保重要视图会显示在施工图文档集的图纸上。

• 确保视图使用一致并且适当的设置。

要创建一视图列表,可以使用 ViewSchedule. CreateViewList 函数。

• ViewSchedule. CreateViewList(Document document)

其中,document 指定创建的明细表所在的文档。

图纸列表是项目中图纸的明细表。图纸列表也可以称为图形索引或图纸索引。还可以将图纸列表用作施工图文档集的目录。通常会将图纸列表放置在标题图纸上。

要创建一视图列表,可以使用 ViewSchedule. CreateSheetList 函数。

• ViewSchedule. CreateSheetList(Document document)

其中,document 指定创建的明细表所在的文档。

配电盘明细表显示有关配电盘、连接到配电盘的线路及其相应负荷的信息。在 Revit® API 里,PanelScheduleView 代表配电盘明细表。要创建一个配电盘明细表,可以通过 PanelScheduleView. CreateInstanceView 函数。

• PanelScheduleView. CreateInstanceView(Document document, ElementId panelId)

其中,document 指定创建的明细表所在的文档,panelId 是配电盘元素。或者

• PanelScheduleView. CreateInstanceView(Document document, ElementId templateId, ElementId panelId)

其中,document 指定创建的明细表所在的文档,templateId 是配电盘模板元素,panelId 是配电盘元素。

第 9 章　事件(Events)

9.1　事件简介

事件是 Revit® 界面或 API 执行过程中触发的特定的动作。通过注册事件，一个插件程序能在某个动作即将发生或刚发生后得到通知，然后根据这个动作采取行动。某些事件是成对的，一个触发在某个动作发生前(称作预备事件“Pre-event”)，另一个则触发在这个动作发生后(称作事后事件“post-event”)。另一些事件非成对出现被叫做“单一”事件。

Revit® 提供了应用级别(如 ApplicationClosing、DocumentOpened)和文档级别(如 DocumentClosing、DocumentPrinting)的事件。Application 类和 ControlledApplication 类都能访问应用级别的事件。ControlledApplication 和 Application 一样代表 Revit® 应用，但 ControlledApplication 无法访问 Revit® 的文档，通常在它的 OnStartup 和 OnShutdown 函数里分别注册和注销事件。这两个类的注册和注销事件的代码是一致的。

事件也可以分为数据库(DB)事件和界面(UI)事件。Application 类和 Document 类都能访问数据库事件，而界面事件只有 UIApplication 能访问(当前，界面事件只有应用级别的)。

一些事件是只读的。这意味着在执行这些事件的时候，模型是无法被修改的。在 API 帮助文档中注明了哪些事件是只读的。另一些常规的事件(非只读)在执行时也有可能无法修改模型，这是因为该模型处在一个无法被修改的状态。程序员可以通过属性 Document. IsModifiable 和 Document. IsReadOnly 来检查文档是否可以被修改。

9.2　注册和注销事件

如何来使用事件？一共需要两个步骤。首先，需要一个函数在得到事件通知后进行响应和处理。这个函数必须有两个参数，第一个参数是 Object 对象表示事件的发送者，第二个参数是事件特定的对象，包含了特定于该事件的参数。例如，要注册 DocumentSavingAs 事件，事件处理函数的第一个参数是 Object 对象，第二个参数是一个 DocumentSavingAsEventArgs 对象，如 void OnDocumentSavingAsEventHandler(Object sender, DocumentSavingAsEventArgs args)。

使用事件的第二步是使用之前定义的处理函数在 Revit® 中注册该事件。这可以在外部程序实现的 IExternalApplication. OnStartup()函数中注册，也可以在 Revit® 启动后的外部命令中注册。虽然事件既可以在外部程序也可以外部命令中注册，不建议在外部命令中注册，除非在同一个外部命令中注册和注销事件。通常用“+=”来注册一个事件。例如，用前面定义的处理函数来注册 DocumentSavingAs 事件，可以这么写：application. DocumentSavingAs += new EventHandler<DocumentSavingAsEventArgs>(OnDocumentSavingAsEventHandler)。

当不需要响应某个已经注册的事件时，可以注销该事件。如果是在 IExternalApplication. OnStartup()函数中注册的事件，通常建议在 IExternalApplication. OnShutdown()函数中注销。当然，也可以在其他任何需要的时候注销。通常都用“-=”

来注销一个事件。例如，注销前面注册过的 DocumentSavingAs 事件，可以这么写：application. DocumentSavingAs－＝ new EventHandler＜DocumentSavingAsEventArgs＞(OnDocumentSavingAsEventHandler)。

代码片段 9-1 注册了一个 DocumentOpened 事件，当这个事件触发时，程序会修改这个打开的项目的地址。最后在程序退出时注销了该事件。

代码片段 9-1：DocumentOpened 事件

```
public class Application_DocumentOpened : IExternalApplication
{
  public IExternalApplication.Result OnStartup(ControlledApplication application)
  {
     try
     {
      //注册事件
      application.DocumentOpened + =  new EventHandler
      < Autodesk. Revit.Events.DocumentOpenedEventArgs> (application_DocumentOpened);
     }
     catch (Exception)
     {
      return Autodesk.Revit.UI.Result.Failed;
     }

     return Autodesk.Revit.UI.Result.Succeeded;
  }

  public IExternalApplication.Result OnShutdown(ControlledApplication application)
  {
     //注销事件
     application.DocumentOpened- =  new EventHandler
      < Autodesk.Revit.Events.DocumentOpenedEventArgs> (application_DocumentOpened);
     return Autodesk.Revit.UI.Result.Succeeded;
  }


  public void application_DocumentOpened(object sender, DocumentOpenedEventArgs args)
  {
     //从事件参数中获得文档对象
     Document doc =  args.Document;

     Transaction transaction =  new Transaction(doc, "Edit Address");
     if (transaction.Start() = =  TransactionStatus.Started)
     {
      doc.ProjectInformation.Address =
         "United States- Massachusetts- Waltham- 1560 Trapelo Road";
      transaction.Commit();
     }
  }
}
```

9.3　可取消的事件

有一些预备事件在其执行真正的行为前是可以被取消的，如 DocumentSaving 事件。使用 Cancellable 属性来判断一个事件是否可以被取消，设置 Cancel 属性为 true 来取消该事件的执行。例如，在一个文件被保存前检查是否符合某些要求，可以响应 DocumentSaving 或 DocumentSavingAs 事件，如果某些要求不符合就取消保存或另存为的行为。预备事件一旦被取消了，就不能反取消。

注意：如果预备事件被取消了，其他注册的事件处理函数不会被触发。但是，相对应的事后事件还是会触发的。

代码片段 9-2 演示了通过响应 DocumentSavingAs 事件的处理函数检查 ProjectInformation 状态参数是否为空。如果是就取消另存行为。注意，如果取消了一个事件，通常要给用户一个解释。

代码片段 9-2：DocumentSavingAs 事件响应函数

```
private void CheckProjectStatusInitial(Object sender, DocumentSavingAsEventArgs args)
{
  Document doc =  args.Document;
  ProjectInfo proInfo =  doc.ProjectInformation;

  //项目信息只存在于项目文档中。
  if (null ! =  proInfo)
  {
     if (string.IsNullOrEmpty(proInfo.Status))
     {

      // 取消另存行为
      args.Cancel =  true;
      MessageBox.Show("项目参数没有设置。取消保存。");
     }
  }
}
```

注意：大部分事件参数有 Cancel 和 Cancellable 属性。但 DocumentChanged 和 FailuresProcessing 使用的是 Cancel()和 IsCancellable()方法。

9.4　数据库事件

表 9-1 列出了所有的数据库事件、它们的类型以及访问级别。

表 9-1　　**DB 事件类型及文档级别列表**

事件	类型	Application 和 ControlledApplication	文档级别
ApplicationInitialized	单一	X	
DocumentChanged	单一	X	
DocumentClosing	预备	X	X

续表

事件	类型	Application 和 ControlledApplication	文档级别
DocumentClosed	事后	X	
DocumentCreating	预备	X	
DocumentCreated	事后	X	
DocumentOpening	预备	X	
DocumentOpened	事后	X	
DocumentPrinting	预备	X	X
DocumentPrinted	事后	X	X
DocumentSaving	预备	X	X
DocumentSaved	事后	X	X
DocumentSavingAs	预备	X	X
DocumentSavedAs	事后	X	X
DocumentSynchronizingWithCentral	预备	X	
DocumentSynchronizedWithCentral	事后	X	
FailuresProcessing	单一	X	
FileExporting	预备	X	
FileExported	事后	X	
FileImporting	预备	X	
FileImported	事后	X	
ProgressChanged	单一	X	
ViewPrinting	预备	X	X
ViewPrinted	事后	X	X

• ApplicationInitialized：当 Revit® 程序被初始化以后，所有的外部程序已经启动，Revit®已经准备好处理文档时触发。通常该事件在外部应用中通过 ControlledApplication 来注册。

• DocumentChanged：当事务被提交、撤销或重做的时候触发，表示文档被修改了。这是个只读事件，用类让外部数据有机会和 Revit® 数据库的状态保持同步。该事件的 DocumentChangedEventArgs 类有几个方法来获得新添加的元素（GetAddElementIds()），被删除的元素（GetDeletedElementIds()）和被修改的元素（GetModifiedElementIds()）。其中 GetAddElementIds() and GetModifiedElementIds() 有重载的方法可以传入一个 ElementFilter 对象来过滤自己感兴趣的那些元素。

• DocumentClosing：当 Revit 要关闭一个文档前触发。

• DocumentClosed：当 Revit 刚关闭完一个文档时触发。

• DocumentCreating：当 Revit 要创建一个新文档时触发。

• DocumentCreated：当 Revit 刚创建完一个文档时触发。

• DocumentOpening：当 Revit 要打开一个文档时触发。

• DocumentOpened：当 Revit 刚打开完一个文档时触发。

- DocumentPrinting：当 Revit 要打印一个视图或视图集时触发。
- DocumentPrinted：当 Revit 刚打印完一个视图或视图集时触发。
- DocumentSaving：当 Revit 要保存一个文档时触发。
- DocumentSaved：当 Revit 刚保存完一个文档时触发。
- DocumentSavingAs：当 Revit 要另存一个文档时触发。
- DocumentSavedAs：当 Revit 刚另存完一个文档时触发。
- DocumentSynchronizingWithCentral：当 Revit 要同步中心模型时触发。
- DocumentSynchronizedWithCentral：当 Revit 刚同步完中心模型时触发。
- FailuresProcessing：当 Revit 在一个事务结束处理失败时触发。
- FileExporting：当 Revit 要导出一个 API 支持的文件格式时触发。
- FileExported：当 Revit 刚导出完一个 API 支持的文件格式时触发。
- FileImporting：当 Revit 要导入一个 API 支持的文件格式时触发。
- FileImported：当 Revit 刚导入完一个 API 支持的文件格式时触发。
- ProgressChanged：当 Revit 里的一个操作有进度条的时候触发。
- ViewPrinting：当 Revit 要打印一个视图时触发。
- ViewPrinted：当 Revit 刚打印完一个视图时触发。

9.5　界面事件

表 9-2 列出了所有的界面事件，它们的类型以及访问级别。

表 9-2　　界面事件类型列表

事件	类型	UIApplication	UIControlledApplication	UIDocument
ApplicationClosing	预备	X	X	
DialogBoxShowing	单一	X	X	
DisplayingOptionsDialog	单一	X	X	
DockableFrameFocusChanged	单一	X	X	
DockableFrameVisibilityChanged	单一	X	X	
Idling	单一	X	X	
ViewActivated	预备	X	X	
ViewActivating	事后	X	X	

- ApplicationClosing：当 Revit 要被关闭的时候触发。
- DialogBoxShowing：当 Revit 要显示一个对话框的时候触发。
- DisplayingOptionsDialog：当 Revit 要显示一个属性对话框的时候触发。
- DockableFrameFocusChanged：当 DockableFrame 的焦点(Focus)发生改变时触发。
- DockableFrameVisibilityChanged：当 DockableFrame 的可见属性发生改变时触发。
- Idling：当 Revit 空闲时(不在处理事务或活动工具)触发。
- ViewActivating：当 Revit 要激活一个视图时触发。
- ViewActivated：当 Revit 刚激活完一个视图时触发。

9.6 闲置事件(IdlingEvent)

1) 闲置事件简介

Revit® API 中的闲置事件(IdlingEvent)提供了另一种与 Revit®程序交互的方式。它的定义如下:public event EventHandler<IdlingEventArgs> UIApplication. Idling。

顾名思义,该事件在 Revit®闲置的时候,也就是用户不对 Revit®进行操作的时候触发。触发的另外一个条件是在该状态下 API 程序可以安全的访问当前文档。在使用某个工具(如画墙),或者某个事务未结束时,该事件是不会被触发的。

闲置事件允许其处理函数开启事务,然后对文档进行修改。该事务会显示在"Undo"列表里,并且可以被用户撤销。

由于闲置事件是在用户界面操作之间被触发的,如果其处理函数需要大量的时间来运行,那么用户会感觉到 Revit®的反应会变慢。所以一般建议处理函数不要做大量的计算,可以把这些计算分散到多次函数处理中,这样用户的体验会大幅提高。

闲置事件有两种模式:默认模式和非默认模式。在默认模式下,每个闲置事件会在每次 Revit®进入闲置状态时触发。如果用户一直在 Revit®界面中活动(Active),那么闲置事件在用户停止移动鼠标或完成一个命令后会一直被触发。如果用户根本不在活动状态,Revit®不会触发更多的闲置事件。这意味着,当用户离开电脑一段时间,这段时间不会触发闲置事件,也就无法利用这些空闲状态。相反,在非默认模式下,Revit®会持续不断的触发闲置事件,即使用户已经停止对电脑的操作。这样的好处是可以充分利用这些空闲时间来做计算,坏处是 CPU 会一直处于满负荷运行状态。可以通过 IdlingEventArgs. SetRaiseWithoutDelay()来将触发模式设置成非默认模式,而且必须在每次闲置事件的处理函数中都调用一下,否则下次触发 Revit®会自动切换到默认模式。

注意:闲置事件是无法被取消的。

2) 示例

下面介绍一个使用闲置事件的小例子。代码片段 9-3 将创建一个文字,并且在闲置事件中每秒更新一次文字内容。

代码片段 9-3:闲置事件响应函数

```
//实现外部程序的 Execute 函数
//创建一个文字,并且注册一个闲置事件
TextNote textNote = null;
String oldDateTime = null;
public Result Execute (ExternalCommandData commandData, ref string message, ElementSet
    elements)
{
  UIApplication uiApp = new UIApplication(commandData.Application.Application);
  Document doc = commandData.Application.ActiveUIDocument.Document;
  using (Transaction t = new Transaction(doc, "Text Note Creation"))
  {
    t.Start();
    textNote = doc.Create.NewTextNote(doc.ActiveView, XYZ.Zero, XYZ.BasisX, XYZ.BasisY,
    0, TextAlignFlags.TEF_ALIGN_LEFT, DateTime.Now.ToString());
```

```
    t.Commit();
  }
  oldDateTime = DateTime.Now.ToString();
  uiApp.Idling += new EventHandler< IdlingEventArgs> (idleUpdate);
  return Result.Succeeded;
}

//闲置事件处理函数
//当时间有更新就将文字内容改为当前时间
public void idleUpdate(object sender, IdlingEventArgs e)
{
  UIApplication uiApp = sender as UIApplication;
  Document doc = uiApp.ActiveUIDocument.Document;
  if (oldDateTime != DateTime.Now.ToString())
  {
    using (Transaction transaction = new Transaction(doc, "Text Note Update"))
    {
     transaction.Start();
     textNote.Text = DateTime.Now.ToString();
     transaction.Commit();
    }
    oldDateTime = DateTime.Now.ToString();
  }
}
```

9.7　外部事件(ExternalEvent)

1) 外部事件简介

Revit® API 提供了外部事件框架可以用于非模态对话框。它是特别针对异步处理所开放的事件接口。它和闲置事件(IdlingEvent)处理类似,但可以由用户自己触发。

要使用外部事件来实现非模态对话框可以通过下面几步来实现:

- 继承并实现实现外部事件处理接口(IExternalEventHandler);
- 用静态方法 ExternalEvent. Create()方法来创建一个外部事件(ExternalEvent);
- 当非模态对话框中某个时候需要调用 Revit®方法时,调用 ExternalEvent. Raise();
- Revit®会在下个闲置时间(idling time cycle)到来时调用 IExternalEventHandler. Execute()方法的实现。

2) ExternalEvent 的创建

调用外部事件类 ExternalEvent. Create()方法来创建一个外部事件。该事件所创建的一个实例会被返回给创建者,创建者使用该实例来向 Revit®发信号触发该事件。Revit®会不时地检查是否有外部事件发信号,如果有就会调用外部事件相应的 Execute()函数。

3) ExternalEvent 的触发

调用外部事件类 ExternalEvent. Raise()方法来触发一个外部事件。当它被调用后,Revit®会等到下一个闲置时间(idling time)来调用注册好的 IExternalEventHandler. Execute()方法。

4) IExternalEventHandler 接口

这是外部事件需要实现的接口。实现这个接口的实例会被注册到 Revit® 中，当每次相应的外部事件触发时，该接口的 Execute()方法会被调用。

IExternalEventHandler 只有两个接口需要实现：Execute()方法和 GetName()方法。后者要返回事件的名字。

代码片段 9-4 展示了一个完整外部事件应用的例子。

首先是实现 IExternalEventHandler 接口，下面代码当事件被调用时显示一个任务对话框。

代码片段 9-4：实现 IExternalEventHandler 接口

```
public class ExternalEventExample : IExternalEventHandler
{
  public void Execute(UIApplication app)
  {
    TaskDialog.Show("External Event", "Click Close to close.");
  }
  public string GetName()
  {
    return "External Event Example";
  }
}
```

接下来是创建外部事件并将上面的事件处理函数注册给外部事件。代码片段 9-5 实现了 IExternalApplication，其中 ShowForm()函数创建一个外部事件和外部事件响应处理的实例，然后显示一个非模态对话框。该非模态对话框有两个按钮，一个来触发事件，一个关闭对话框。

代码片段 9-5：外部事件的注册和触发

```
public class ExternalEventExampleApp : IExternalApplication
{
  public static ExternalEventExampleApp thisApp =  null;
  //非模态对话框实例
  private ExternalEventExampleDialog m_MyForm;

  public Result OnShutdown(UIControlledApplication application)
  {
    if (m_MyForm ! =  null && m_MyForm.Visible)
    {
     m_MyForm.Close();
    }

    return Result.Succeeded;
  }

  public Result OnStartup(UIControlledApplication application)
  {
```

```
      m_MyForm = null;   //在外部命令中创建非模态对话框
      thisApp = this;  //静态变量,保存 Application 实例

      return Result.Succeeded;
    }

    //外部命令调用此方法
    public void ShowForm(UIApplication uiapp)
    {
      //如果没有创建对话框,创建并显示它
      if (m_MyForm == null || m_MyForm.IsDisposed)
      {
        // 新建一个外部事件响应实例
        ExternalEventExample handler = new ExternalEventExample();

        // 新建一个外部事件实例
        ExternalEvent exEvent = ExternalEvent.Create(handler);

        // 把上面两个实例传给对话框.
        m_MyForm = new ExternalEventExampleDialog(exEvent, handler);
        m_MyForm.Show();
      }
    }
}

[Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMode.
Manual)]
[Autodesk.Revit.Attributes.Regeneration(Autodesk.Revit.Attributes.RegenerationOption.
Manual)]
public class Command : IExternalCommand
{
  public virtual Result Execute(ExternalCommandData commandData, ref string message,
  ElementSet elements)
  {
    try
    {
      ExternalEventExampleApp.thisApp.ShowForm(commandData.Application);
      return Result.Succeeded;
    }
    catch (Exception ex)
    {
      message = ex.Message;
      return Result.Failed;
    }
  }
}

public partial class ExternalEventExampleDialog : Form
{
  private ExternalEvent m_ExEvent;
  private ExternalEventExample m_Handler;
```

```
    public ExternalEventExampleDialog(ExternalEvent exEvent, ExternalEventExample handler)
    {
        InitializeComponent();
        m_ExEvent = exEvent;
        m_Handler = handler;
    }

    protected override void OnFormClosed(FormClosedEventArgs e)
    {
        //保存的实例需要释放
        m_ExEvent.Dispose();
        m_ExEvent = null;
        m_Handler = null;

        base.OnFormClosed(e);
    }

    private void closeButton_Click(object sender, EventArgs e)
    {
Close();
    }
    //按钮响应函数,点击按钮触发外部事件
    private void showMessageButton_Click(object sender, EventArgs e)
    {
        m_ExEvent.Raise();
    }
}
```

第 10 章　功能区扩展(Ribbon UI)

10.1　基础介绍

要了解 Revit® API 对 Ribbon UI 的支持，首先需要有如下几点的认识。

• Revit®允许用户通过 API 订制自己的 Ribbon 控件，相关的接口在 Revit® APIUI. dll，用户需要在自己的应用程序引用这个 dll。

• 实现自定义控件的时机是，Revit®启动时创建自定义控件。

• 目前 Revit® API 提供了以下几种基本控件：选项卡页(RibbonTab)，面板(RibbonPanel)，命令按钮(PushButton)，下拉按钮(PulldownButton)，下拉切换按钮(SplitButton)，下拉组合框(Combobox)，切换按钮(RadioButtonGroupand ToggleButton)，文本框(TextBox)和分隔符(Separator)。

• 其中 RibbonTab 和 RibbonPanel 是容器类控件，是用来管理、放置具体控件。容器和具体的控件关系为，控件必须放在 RibbonPanel 内，RibbonPanel 必须放在 RibbonTab 内。

• 用户可以创建自己的选项卡页来组织自己的控件；也可以在把自己的控件添加到 Revit®主程序的选项卡页(目前仅限于 Analyze 选项卡页和 Add-Ins 选项卡页)。

1) API 类

除容器类控件(RibbonTab 和 RibbonPanel)，每种 Ribbon 控件都有两个 API 类与它相关。一个是继承自 RibbonItemData 的配置类，用来配置创建控件所需的数据。另一个是继承自 RibbonItem 的控件类(用来表示已经创建好的控件)。配置类的所有属性在控件类也同样存在。用户可以在创建控件前，通过配置类来设置这些属性，也可以在创建好控件后，通过控件类的属性来修改或补充这些信息。

2) 控件弹出信息

大部分的控件都可以通过 ToolTip 属性设置简单的弹出信息。也可以通过设置 LongDescription 和 ToolTipImage 属性来扩展弹出信息。当把鼠标放在控件上时，这些信息将被显示。见图 10-1。

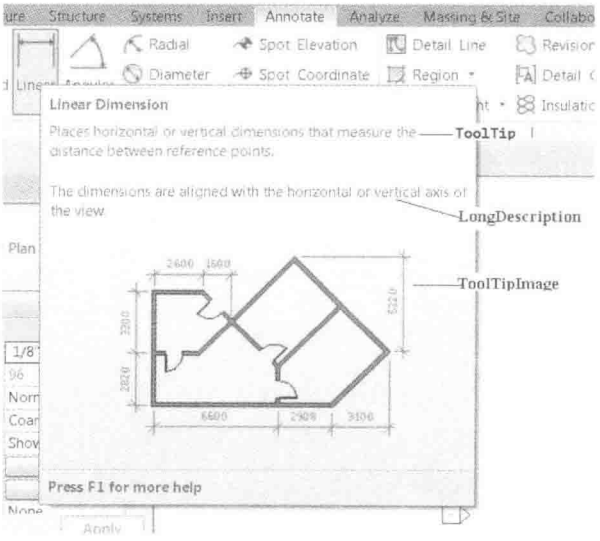


图 10-1　弹出信息

3) 控件相关联图片

按钮类控件都可以通过 LargeImage 和 Image 属性设置相关联的的图片。层叠布局的控件和小控件(如文本框、

组合框等)只有 Image 属性。LargeImage 最适合的图片是 32×32 像素,它会被显示在控件上。Image 最适合的图片是 16×16 像素,当控件以层叠样布局或者放到快速访问工具栏上时,图片会被显示在控件上。

4) 控件的可见性和可用性

通过 RibbonItem. Visible 属性来隐藏/显示控件。通过 RibbonItem. Enabled 属性来控制控件可用/不可用。

5) 编程如何实现

创建工程,引用 Revit APIUI. dll;新建一个类,继承 IExternalApplication 接口;然后实现 OnStartup 和 OnShutdown 函数,在 OnStartup 函数里创建 Ribbon 控件。

代码片段 10-1 是以创建一个文本框为例。

代码片段 10-1:创建一个文本框

```
//继承 IExternalApplication 接口
public class Ribbon : Autodesk.Revit.DB.IExternalDBApplication
{
   ///实现 OnStartup
   public Autodesk.Revit.UI.Result OnStartup(UIControlledApplication application)
   {
      //在 OnStartup 函数里创建一个 Ribbon 文本框
      application.CreateRibbonTab("CustomTag");
      RibbonPanel panel =  application.CreateRibbonPanel("CustomTag", "CustomPanel");
      TextBox textBox =  panel.AddItem(new TextBoxData("CustomTextBox")) as TextBox;

      return Autodesk.Revit.UI.Result.Succeeded;
   }
   //实现 OnStartup
   public Autodesk.Revit.UI.Result OnShutdown(UIControlledApplication application)
   {
      return Autodesk.Revit.UI.Result.Succeeded;
   }
}
```

10.2 选项卡页(RibbonTab)

用户可以创建一个新的选项卡页,用来管理自己的面板 RibbonPanel。

- voidUIControlledApplication. CreateRibbonTab(string tabName)

10.3 面板(RibbonPanel)

面板是其他控件的容器。当用户要以 API 来放置排列一组自己的控件、隐藏/显示、可使用/不可用一组时,RibbonPanel 类就尤其有用。

1) 创建面板

创建面板有三个重载函数,通过这三个函数可以分别实现在客户自定义的选项卡页,Add-Ins 选项卡页或者 Analyze 选项卡页上创建面板。

• RibbonPanelUIControlledApplication. CreateRibbonPanel(string tabName, string panelName)

在指定的选项卡页上创建一个面板,tabName 只能是用户自己创建的选项卡页的名字。

这里的 tabName 是 CreateRibbonTab 时传递的 tabName。

• RibbonPanelUIControlledApplication. CreateRibbonPanel(string panelName)

在 Add-Ins 选项卡页上创建一个面板。

• RibbonPanelUIControlledApplication. CreateRibbonPanel(Tab tab, string panelName)

在 Revit® 自身的选项卡页上创建一个面板,目前只开放了仅限于 Analyze 选项卡页和 Add-Ins 选项卡页。

2) 找回用户创建的面板

• List<RibbonPanel>UIControlledApplication. GetRibbonPanels()

返回 Add-Ins 选项卡页上的通过 API 创建的所有面板。

• UIControlledApplication. GetRibbonPanels(Autodesk. Revit. UI. Tab)

返回 Analyze 选项卡页或 Add-Ins 选项卡页上的通过 API 创建的所有面板。

• UIControlledApplication. GetRibbonPanels(string)

返回指定的选项卡页上的通过 API 创建的所有面板。

注意:

① 这里的面板必须是用户通过 API 创建的。

② GetRibbonPanels 只能拿到通过 API 创建面板,不能拿到系统的面板。关于如何拿到系统的面板,目前 Revit® 没有提供相应的 API。

3) 往面板添加控件

在面板上添加自己的控件有三个重载函数:

• RibbonItemRibbonPanel. AddItem(RibbonItemData itemData)

用户可以通过此方法在 panel 上创建各种控件,如文本框、下拉按钮、组合框等。具体如何创建会在不同控件相应的章节详细介绍。

此方法创建的控件在面板上是默认布局,是大尺寸控件独立排列在面板的一列,如图 10-2 的前三个控件。

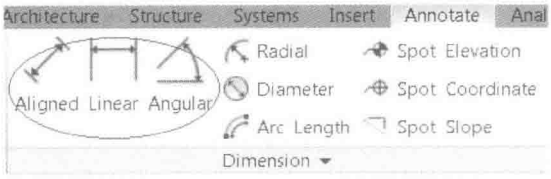


图 10-2　控件默认布局



图 10-3　控件层叠布局

• IList< RibbonItem >RibbonPanel. AddStackedItems(RibbonItemData item1, RibbonItemData item2)

在面板上添加两个层叠布局的控件,这两个控件以小尺寸的方式,层叠在面板的同一列

上，如图 10-3 所示。

- IList < RibbonItem > RibbonPanel. AddStackedItems (RibbonItemData item1, RibbonItemData item2, RibbonItemData item3)

在面板上添加三个层叠布局的控件，这三个控件以小尺寸的方式，水平层叠在面板的同一列上，如图 10-4 所示。

所有的控件都支持通过 AddItem 创建成普通样式，PushButton，PulldownButton，ComboBox 和 TextBox 支持通过 AddStackedItems 层叠布局。RadioButtonGroup 和 SplitButton 不支持层叠布局。

注意：在后续章节关于各个具体控件的创建部分，都只提到创建普通样式的控件。创建层叠布局控件可以以此类推，换成使用 AddStackedItems 方法即可，就不再具体说明。

4）添加分割符

- voidRibbonPanel. AddSeparator()

在 panel 上添加竖直的分隔线，分割线将被显示在上一次添加的控件的右边，后续再添加的控件将显示在此分隔线的右边。

可以添加多条分隔线。如果没有控件被添加在面板上，添加的分隔线也将不被显示(图 10-5)。

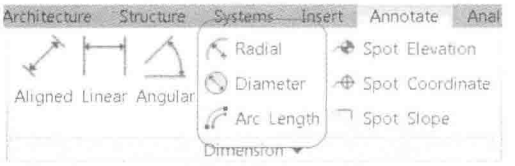


图 10-4　控件层叠布局

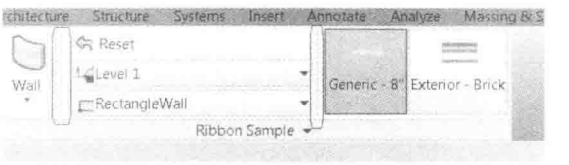


图 10-5　Panel 上的分割线

5）通过面板控制一组控件的隐藏/显示

- boolRibbonPanel. Visible

6）通过面板控制一组控件的可用性

- boolRibbonPanel. Enabled

10.4　命令按钮(PushButton)

命令按钮是一个单一命令按钮。它对应一个 ExternalCommand，单击这个按钮，对应的 ExternalCommand 将被执行。见图 10-6。

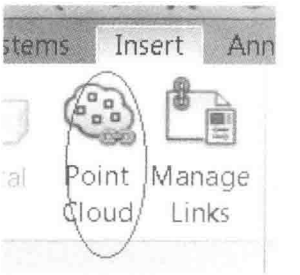


图 10-6　命令按钮

创建命令按钮：

- RibbonItemRibbonPanel. AddItem (RibbonItemData itemData)

用于创建普通样式的命令按钮。这里的 RibbonItemData 需要传递一个 PushButtonData，配置类里必须包含以下信息：控件的名字，显示在控件上的文本以及对应的 ExternalCommand (由 assemblyName 和 className 来指定)。代码片段 10-2 展示的是如何创建一个命令按钮。

代码片段 10-2:创建一个命令按钮

```
PushButtonData pushButtonData =  new PushButtonData (" name", " Text", ECAssemblyPath,
ECFullName);
PushButton pushButton =  panel.AddItem(pushButtonData) as PushButton;
```

10.5　下拉按钮(PulldownButton)

下拉按钮是带有下拉列表的按钮控件,下拉列表中的每个元素都是一个命令按钮,在下拉列表的元素间可以加水平分割线。见图 10-7。

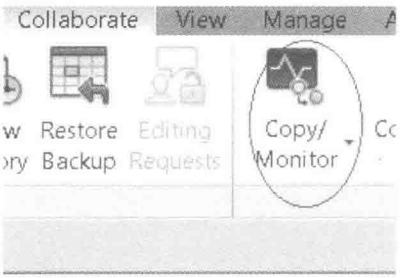


图 10-7　下拉按钮

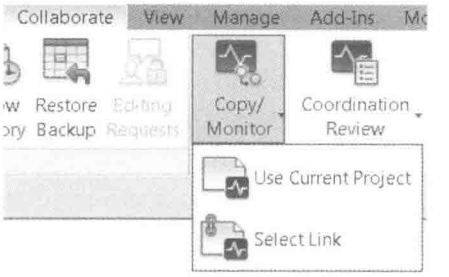


图 10-8　展开下拉按钮

下拉按钮的右侧显示有一个向下的箭头,单击这个箭头或者单击整个下拉按钮,展开整个列表;然后单击列表中的某一个命令按钮,将执行对应的 ExternalCommand。见图 10-8。

可能过以下方式创建下拉按钮:

- RibbonItemRibbonPanel. AddItem(RibbonItemData itemData)

创建普通样式的下拉按钮。这里的 RibbonItemData 需要传递一个 PulldownButtonData,配置类里必须包含控件的名字和显示在控件上的文本。

- PushButtonPulldownButton. AddPushButton(PushButtonData buttonData)

往下拉按钮的下拉列表里添加命令按钮,如果下拉列表没有元素,这个下拉按钮将不会被显示。

- voidPulldownButton. AddSeparator()

往下拉按钮的下拉列表里添加水平分割线,分割线将紧接着最近一次添加的命令按钮显示。后边继续添加的命令按钮将显示在分割线下边。可以添加多条水平分割线。

10.6　下拉记忆按钮(SplitButton)

下拉记忆按钮是下拉按钮的派生类控件,它和下拉按钮非常类似。不同的是下拉记忆按钮是在正下方有一个向下的箭头(图 10-9、图 10-10),而下拉按钮的是在右侧;更重要的区别下拉记忆按钮会和它下拉列表里的某一个命令按钮相关联。单击下拉记忆按钮或这个箭头,将执行被关联的命令按钮所对应的 ExternalCommand。

由于下拉记忆按钮是下拉按钮的派生类控件,下拉按钮所有的 API 功能下拉记忆按钮也有。这里不再一一重述。本节将着重讲解子下拉记忆按钮与下拉列表中的命令按钮相关联。

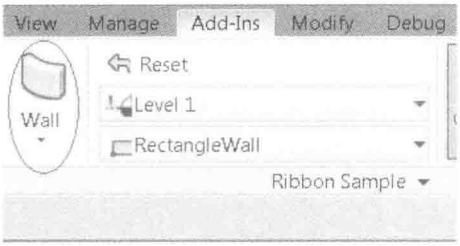


图 10-9　下拉记忆按钮

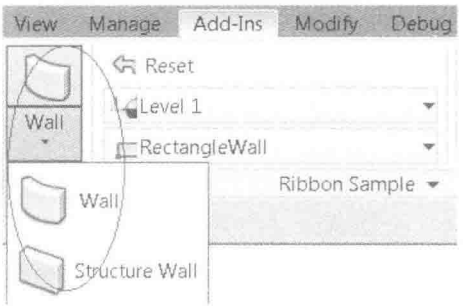


图 10-10　展开下拉记忆按钮列表

① 两个相关的 API 函数：

• PushButtonSplitButton. CurrentButton

获取或设置与下拉记忆按钮相关联的命令按钮，这个命令按钮必须是本下拉记忆按钮的下拉列表中的一员。

• boolSplitButton. IsSynchronizedWithCurrentItem

是否可以设置下拉记忆按钮的当前按钮 CurrentButton。

② 下拉按钮与它下拉列中的一个命令按钮相关联，这个被关联的命令按钮就叫当前按钮。下拉记忆按钮显示当前按钮的按钮图片，按钮弹出信息，按钮文本等；单击下拉记忆按钮也将执行当前按钮对应的 ExternalCommand。

③ 当前按钮的默认值为记忆下拉按钮的下拉列表中的第一个命令按钮，IsSynchronizedWithCurrentItem 的默认值为 true。

④ 当 IsSynchronizedWithCurrentItem 为 true 时，用户在 Revit® UI 单击下拉列表中的命令按钮，当前按钮将被自动同步这个命令按钮，用户也可以通过 CurrentButton 属性来设置被关联的命令按钮。

⑤ 当 IsSynchronizedWithCurrentItem 为 false 时，当前按钮将始终为 SplitButton 下拉列表中的第一个可用的命令按钮，CurrentButton 属性不能被设置。

代码片段 10-3 展示的是如何创建一个包含两个子命令按钮的下拉切换按钮。

代码片段 10-3：创建一个包含两个子命令按钮的下拉切换按钮

```
# region 创建一个包含两个 pushButton 的 SplitButton，用来创建结构墙和非结构墙
//创建一个 SplitButton
SplitButtonData splitButtonData =  new SplitButtonData("NewWallSplit", "Create Wall");
SplitButton splitButton         =  panel.AddItem(splitButtonData) as SplitButton;

//创建一个 pushButton 加到 SplitButton 的下拉列表里
PushButton pushButton =  splitButton.AddPushButton(new PushButtonData("WallPush", "Wall",
                         AddInFullPath, "Revit.SDK.Samples.Ribbon.CS.CreateWall"));
pushButton.LargeImage =   new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder, "
                          CreateWall.png"), UriKind.Absolute));
pushButton.Image      =   new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder, "
                          CreateWall-S.png"), UriKind.Absolute));
```

```
pushButton.ToolTip        = "Creates a partition wall in the building model.";
pushButton.ToolTipImage   =  new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder, "
                             CreateWallTooltip.bmp"), UriKind.Absolute));

//创建另一个 pushButton 加到 SplitButton 的下拉列表里
pushButton                = splitButton.AddPushButton(new PushButtonData("StrWallPush", "
                            Structure Wall", AddInPath, "Revit.SDK.Samples.Ribbon.CS.
                            CreateStructureWall"));
pushButton.LargeImage     =  new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder, "
                             StrcturalWall.png"), UriKind.Absolute));
pushButton.Image          =  new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder, "
                             StrcturalWall-S.png"), UriKind.Absolute));
# endregion
```

10.7　下拉组合框(ComboBox)

下拉组合框是有下拉列表的选择控件，通过单击控件上的箭头可显示或隐藏下拉列表，此控件不支持多项选择。见图 10-11。

① 创建下拉组合框：

• RibbonItemRibbonPanel.AddItem(RibbonItemData itemData)

这里的 RibbonItemData 需要传递一个 ComboBoxData。

② ComboBox 类的重要成员：

• ComboBoxMemberComboBox.Current

③ 获取或设置当前选择项：

• voidComboBox.AddSeparator()

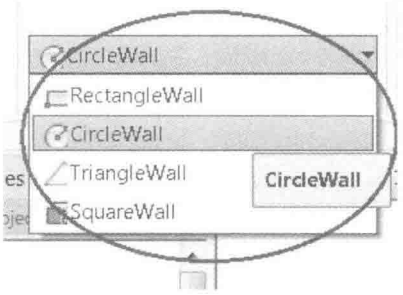


图 10-11　下拉组合框

④ 下拉组合框支持往它的下拉列表的可选项间添加分隔符。

• CurrentChanged 事件

当改变当前的选择项时触发此事件。

• DropDownOpened 事件

当单击组合框上的箭头显示下拉列表时触发此事件。

• DropDownClosed 事件

当单击组合框上的箭头隐藏下拉列表时触发此事件。

10.8　下拉组合框的可选项(ComboBoxMember)

向下拉组合框中添加可选项有两个重载函数：

• ComboBoxMemberComboBox.AddItem(ComboBoxMemberData memberData)

• IList<ComboBoxMember>ComboBox.AddItems(IList<ComboBoxMemberData memberData>)

可选项是可以被分组的，在创建可选项时通过指定 ComboBoxMemberData 的

GroupName 来实现。代码片段 10-4 展示的是如何创建一个下拉组合框来选择墙的类型。

代码片段 10-4:创建一个下拉组合框来选择墙的类型

```
// Prepare data to create ComboBox
ComboBoxData comboBoxData = new ComboBoxData("WallShapeComboBox");
// Create ComboBox
ComboBox comboboxWallShape = panel.AddItem(comboBoxData) as ComboBox;

//Add options to WallShapeComboBox
// Prepare data to create ComboBoxMember
ComboBoxMemberDataboxMemberData = new ComboBoxMemberData ("RectangleWall","
    RectangleWall");
// CreateComboBoxMember
ComboBoxMember boxMember = comboboxWallShape.AddItem(boxMemberData);
boxMember.Image = new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder, "RectangleWall.
    png"), UriKind.Absolute));

// Prepare data to create ComboBoxMember
boxMemberData = new ComboBoxMemberData("CircleWall", "CircleWall");
// CreateComboBoxMember
boxMember = comboboxWallShape.AddItem(boxMemberData);
boxMember.Image = new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder, "CircleWall.
    png"), UriKind.Absolute));

// Prepare data to create ComboBoxMember
boxMemberData = new ComboBoxMemberData("TriangleWall", "TriangleWall");
// CreateComboBoxMember
boxMember = comboboxWallShape.AddItem(boxMemberData);
boxMember.Image = new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder, "TriangleWall.
    png"), UriKind.Absolute));

// Prepare data to create ComboBoxMember
boxMemberData = new ComboBoxMemberData("SquareWall", "SquareWall");
// CreateComboBoxMember
boxMember = comboboxWallShape.AddItem(boxMemberData);
boxMember.Image = new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder, "SquareWall.
    png"), UriKind.Absolute));
```

10.9 选择按钮组和切换按钮(RadioButtonGroup/ToggleButton)

如图 10-12 所示的是选择按钮组和它的成员切换按钮。

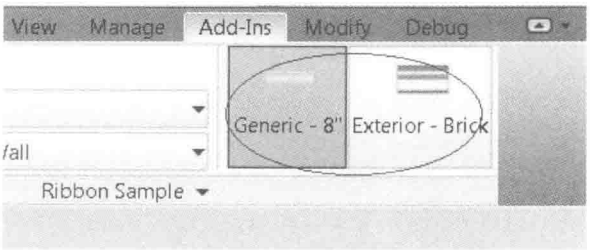


图 10-12 选择按钮组和它的成员切换按钮

1) 选择按钮组

选择按钮组是一组互斥的切换按钮。用户一次只能选择选择按钮组中的一个切换按钮,通过 RadioButtonGroup.Current 属性获取或设置当前被

选择的切换按钮。

2）切换按钮

切换按钮是一个用户可选择的按钮，它被作为选择按钮组中的一个项目来使用。在选中某个切换按钮后，无法通过单击它来清除选择。

ToggleButton 继承自 PushButton，需要注意的是切换按钮既可以对应一个 ExternalCommand；也可以不对应 ExternalCommand，仅作为一个选择控件使用。当切换按钮对应 ExternalCommand，用户在图形界面选择这个切换按钮时，会执行它对应的 ExternalCommand；而通过 RadioButtonGroup. Current 属性来设置当前被选择的切换按钮，则不会执行对应的 ExternalCommand，详见代码片段 10-5 和代码片段 10-6。

代码片段 10-5：创建一个包含两个切换按钮的切换按钮组，其中每一个切换按钮都不执行具体的 ExternalCommand，仅作为选择控件使用

```
//创建一个 RadioButtonGroup
RadioButtonGroupData radioButtonGroupData =  new RadioButtonGroupData("
                                             WallType Selector");
RadioButtonGroup radioButtonGroup         =  (RadioButtonGroup)
                                             (panel.AddItem(radioButton GroupData));

//给 RadioButtonGroup 添加 toggleButton
ToggleButton toggleButton =  radioButtonGroup.AddItem(
new ToggleButtonData("Generic8", "Generic-8\""));
toggleButton.LargeImage   =  new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder,
"Generic8.png"), UriKind.Absolute));
toggleButton.Image        =  new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder,
"Generic8-S.png"), UriKind.Absolute));

//给 RadioButtonGroup 添加 toggleButton
toggleButton              =  radioButtonGroup.AddItem(
new ToggleButtonData("ExteriorBrick", "Exterior-Brick"));
toggleButton.LargeImage=  new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder,
"ExteriorBrick.png"), UriKind.Absolute));
toggleButton.Image        =  new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder,
"ExteriorBrick-S.png"), UriKind.Absolute));
```

代码片段 10-6：创建一个包含两个切换按钮的切换按钮组，其中每一个切换按钮都会执行具体的 ExternalCommand

```
//创建一个 RadioButtonGroup
RadioButtonGroupData radioButtonGroupData =  new RadioButtonGroupData("
WallType Selector");
RadioButtonGroup radioButtonGroup =  (RadioButtonGroup)(panel.AddItem(
radioButton GroupData));

//给 RadioButtonGroup 添加 toggleButton
ToggleButton toggleButton =  radioButtonGroup.AddItem(new ToggleButtonData("Generic8",
"Generic-8\"",
```

```
AddInPath, "Revit.SDK.Samples.Ribbon.CS.CreateGeneric8Wall"));
toggleButton.LargeImage = new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder,
"Generic8.png"), UriKind.Absolute));
toggleButton.Image = new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder,
"Generic8-S.png"), UriKind.Absolute));

//给 RadioButtonGroup 添加 toggleButton
toggleButton = radioButtonGroup.AddItem(new ToggleButtonData("ExteriorBrick",
"Exterior-Brick",
AddInPath, "Revit.SDK.Samples.Ribbon.CS.CreateExteriorBrickWall "));
toggleButton.LargeImage = new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder,
"ExteriorBrick.png"), UriKind.Absolute));
toggleButton.Image = new BitmapImage(new Uri(Path.Combine(ButtonIconsFolder,
"ExteriorBrick-S.png"), UriKind.Absolute));
```

10.10 文本框(TextBox)

文本框是文本输入控件,如图 10-13 所示。

1) 创建文本框

• RibbonItemRibbonPanel. AddItem (RibbonItemData itemData)

创建文本框,这里的 RibbonItemData 需要传递一个 TextBoxData,它必须包含文本框的名字信息。

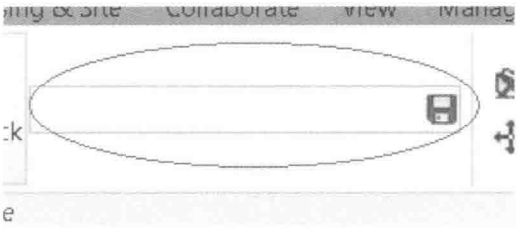


图 10-13 文本框

2) 文本框类的重要成员

• boolShowImageAsButton

文本框的 Image 可以被显示成一个可点击的按钮。当设置 ShowImageAsButton 的值为 true 时,Image 一按钮样式显示在文本框的右边。用户单击这个 Image 按钮,等同于用户单击 Enter 键。当设置 ShowImageAsButton 的值为 false 时,image 以图片显示在文本框的左边。ShowImageAsButton 的默认值是 false。

• stringPromptText

除了与其他基本控件一样,可以提供弹出式的帮助信息,文本框还可以通过设置 PromptText 属性来设置一些提示文本,向用户指明该文本框应该输入什么类型的信息。当文本框里没有值并且没有获得输入焦点的时候,提示文本以斜体字显示在文本框中。如图 10-14 所示,该文本框的提示文本为“new wall”。

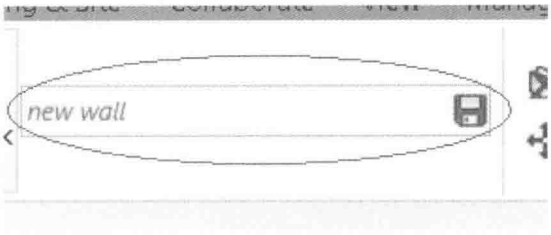


图 10-14 文本框的提示文本

• boolSelectTextOnFocus

SelectTextOnFocus 为 true 时,当文本框获得输入焦点,文本框中的值将被选中;否者就不被选中。SelectTextOnFocus 的默认值是 false。

• ObjectValue

需要注意的是,对文本框而言,从父类

RibbonItem 继承来的 Text 属性是无效的。文本框自己提供了 Value 属性来得到或设置文本框内输入的值。

文本框的值可以是 string 类型也可以是其它任何数据类型。如果 Value 的类型不是 string,用户需要自己提供 Value 和 string 之间类型转换方法。默认情况 TextBox 按照 Value. ToString () 来显示。

当用户在图形界面编辑文本框的值时,只有当用户按下 Enter 键或者单击 Image 按钮,文本框的新值才会被接受;否则一旦文本框失去输入焦点,文本框值将恢复到本次编辑之前的值。

- double Width

文本框可以通过 Width 来获取或设置宽度。

- EnterPressed 事件

当用户按下 Enter 键或单击 Image 按钮时触发此事件,见代码片段 10-7。

代码片段 10-7:EnterPresseds 事件 handler

```
voidSetTextValue(object sender, TextBoxEnterPressedEventArgs args)
{
string strText =  m_textBox.Value as string;
}
```

3) 扩展面板(SlideOut)

通过 RibbonPanel. AddSlideOut()方法,可以在当前面板的底部扩展向下滑出的部分。扩展了向下滑出部分的面板,在它的下端有一个向下的箭头(图 10-15),单击此箭头,即可展开下滑部分。此时下滑部分的左下端显示有一个图钉(图 10-16)。单击这个图钉可以使下滑部分订住不再收拢;或者取消钉住,下滑部分收拢。



图 10-15　扩展面板——收拢下滑部分



图 10-16　扩展面板——展开下滑部分

需要注意的是,一旦 RibbonPanel. AddSlideOut()方法被调用,后续往此面板上添加的

控件都将被放置在下滑出部分。在同一个面板上，AddSlideOut()方法只能被调用一次，否者产生 InvalidOperationException 异常。

10.11 Revit®风格的任务对话框(TaskDialog)

在 Revit®中，任务对话框是具有 Revit®风格的 Windows 对话框的替代品，是一种典型的模态对话框。任务对话框拥有一套已经排列好的通用组件设置，这样可以保证和 Revit®界面的一致性。

任务对话框可以被用于：

- 给用户提供信息。
- 询问用户，并得到反馈。
- 允许用户选择，并执行相应命令。

任务对话框有图 10-17 所示的控件，用户并不需要在自定义的任务对话框中实现全部的控件。



图 10-17 任务对话框及其控件

10.11.1 任务对话框的控件

本节将详细介绍任务对话框的每个控件及 API 的使用方法。

1) 标题(Title)

所有的任务对话框都需要标题。任务对话框的标题需要是唯一的，且具描述性的。它遵循以下格式：<插件名称>—<短标题>。

其中：

- <插件名称>指的是触发任务对话框的外部插件名称，这个名称定义在. addin 文件里面。如果没有已经定义的插件名称，则不显示插件名称。
- <短标题>是用户通过 Title 属性赋值的字符串。

用户可以通过 TaskDialg. Title 属性访问和设置标题。也可以在构造任务对话框时，将标题作为参数传入。

2) 主说明(Main Instructions)

主说明将被用大字体显示在任务对话框的最顶端。

- 每个任务对话框都必须有主说明。

• 主说明的内容不得超过 3 行。

• 用户可以通过 TaskDialog. MainInstructions 属性访问和设置主说明。也可以在使用静态函数构造认为对话框时,作为参数传入。

3) 主要内容(Main Content)

主要内容将被用小一号的字体显示在主说明的下方。

• 主要内容不是必须的。它一般用于主说明不足以描述用户设计任务对话框想要表达的内容时,用户可以使用主要内容来作描述。

• 主要内容应该不仅仅是主说明的重申,它应该在主说明的基础上提供更多的信息。

• 用户可以通过 TaskDialog. Maincontent 来访问和设置主要内容。

4) 扩展内容(Expanded Content)

这部分内容默认是被隐藏起来的。如果用户点击"Show Details"按钮,它将被显示在主要内容的下方。

• 扩展内容不是必须的,而且也很少被使用。它一般用于描述那些不是必要的,或者并不适用于大多数情况下的内容。

• 如果用户设置了扩展内容,那么 Show/Hide Details 按钮将出现在任务对话框上。

• 用户可以通过 TaskDialog. ExpandedContent 来访问和设置扩展内容。

5) 命令链接(Command Links)

在 Revit® 任务对话框中,命令链接一般在以下场景被使用:

• 选项为 A, B 或者 C。

• 选项为 A 或者 B 或者 A 和 B。

• 选项为 A 或者非 A。

命令链接文本分为下面两个部分:

• 主要内容(Main content):这部分对于命令链接来说是必须的。

• 补充内容(Supporting content):这部分提供进一步的信息来对主要内容进行说明,可以为空。

一般来说,第一条命令链接会作为当前任务对话框的默认操作。它应该相对其他命令链接而言是一个很普通的命令,或者说至少在用户默认执行该操作时,不会有潜在风险。

用户可以通过下面两个方法来添加命令链接:

• AddCommandLink(CommandLinkId id, string mainContent);

• AddCommandLink (CommandLinkId id, string mainContent, string supportingContent)。

其中 CommandLinkId 是一个有四个枚举值的枚举,如下所示:

• TaskDialogCommandLinkId. CommandLink1;

• TaskDialogCommandLinkId. CommandLink2;

• TaskDialogCommandLinkId. CommandLink3;

• TaskDialogCommandLinkId. CommandLink4。

这说明用户在一个任务对话框上,最多可以添加四个命令链接。

6) 普通按钮 Common Buttons

普通按钮是一类简单的按钮,出现在任务对话框的右下角。Revit® API 提供了一个 TaskDialogCommonButtons 枚举,如下所示。

- TaskDialogCommonButtons. OK;
- TaskDialogCommonButtons. Cancel;
- TaskDialogCommonButtons. Yes;
- TaskDialogCommonButtons. No;
- TaskDialogCommonButtons. Retry;
- TaskDialogCommonButtons. Close;
- TaskDialogCommonButtons. None。

用户可以从中选择自己需要的按钮，赋值给 TaskDialog. CommonButtons 属性，从而定制任务对话框的普通按钮。需要注意的是，Revit® 任务对话框不允许用户自定义普通按钮上的文字。

7）默认按钮或链接 Default button or link

所有的任务对话框都需要有一个默认按钮或者命令链接。如果一个任务对话框有 OK 按钮，那么如果用户不设置默认按钮，则默认为 OK 按钮。用户可以通过 TaskDialog. DefaultButton 属性设置默认按钮或链接。

8）不再提示复选框"Do not show me again" (DNSM) Checkbox

Revit® 任务对话框支持 DNSM 复选框以便用户选择下次是否还显示该对话框。用户可以通过设置 TaskDialog. VerificationText 属性来自定义任务对话框的 DNSM 复选框。如果该属性没有被设值，那么该任务对话框默认没有 DNSM 复选框。用户可以通过 TaskDialog. WasVerificatinChecked()方法来取到用户的反馈。

9）文字信息 Footer Text (optional)

在 Revit® 任务对话框中，文字信息一般用于链接到一个网址，以便用户获取更多的帮助信息。用户可以通过 TaskDialog. FooterText 来访问和设置文字信息。

10.11.2 创建和显示自定义任务对话框

Revit® API 提供了两种方式让用户创建和显示自定义任务对话框。

- 使用 TaskDialog 类的构造函数创建任务对话框，然后使用 Show()方法来显示给用户。
- 使用 TaskDialog. Show()一些静态方法来创建和显示任务对话框。这种方式只适用于创建简单的任务对话框。如果需要自定义复杂对话框，请使用第一种方式。

代码片段 10-8 演示如何通过 Revit® API 创建和显示任务对话框。

代码片段 10-8：外部命令中 Excute 函数的定义

```
//显示任务对话框，并取得返回值。
            TaskDialogResult tResult =  mainDialog.Show();

            //使用对话框返回值来做进一步的事情
            if (TaskDialogResult.CommandLink1 = =  tResult)
            {
                TaskDialog dialog_CommandLink1 =  new TaskDialog("版本信息");
                dialog_CommandLink1.MainInstruction =
                    "版本名:" +  app.VersionName +  "\n"
                    +  "版本号:" +  app.VersionNumber;
```

```
            dialog_CommandLink1.Show();

        }

        else if (TaskDialogResult.CommandLink2 = = tResult)
        {
//使用静态方法创建显示任务对话框
            TaskDialog.Show("活动文档信息",
                "活动文档标题:" + activeDoc.Title + "\n"
                + "活动文档名:" + activeDoc.ActiveView.Name);
        }

        return Autodesk.Revit.UI.Result.Succeeded;
    }

public Autodesk.Revit.UI.Result Execute(ExternalCommandData commandData,
        ref string message, Autodesk.Revit.DB.ElementSet elements)
    {
        Application app = commandData.Application.Application;
        Document activeDoc = commandData.Application.ActiveUIDocument.Document;

        //通过构造函数创建一个 Revit 任务对话框
        TaskDialog mainDialog = new TaskDialog("自定义标题");
        mainDialog.MainInstruction = "任务对话框使用说明:";
        mainDialog.MainContent =
            "这个例子演示如何通过 Revit API 自定义 Revit 样式任务对话框。";
        mainDialog.ExpandedContent = "在 Revit 中,任务对话框可以用于显示信息和接受简
        单输入。";

        //为任务对话框添加命令链接
        mainDialog.AddCommandLink(TaskDialogCommandLinkId.CommandLink1,
                                  "查看 Revit Application 信息");
        mainDialog.AddCommandLink(TaskDialogCommandLinkId.CommandLink2,
                                  "查看当前文档信息");

        //设置普通按钮以及默认按钮
        mainDialog.CommonButtons = TaskDialogCommonButtons.Ok | TaskDialogCommonButtons.
        Cancel;
        mainDialog.DefaultButton = TaskDialogResult.Ok;

        mainDialog.VerificationText = "不再提示该消息";
        //设置文字信息,一般为一个链接
        mainDialog.FooterText =
            "< a href= \"http://www.autodesk.com\"> "
            + "点击此处了解更多< /a> ";
```

10.11.3　任务对话框执行结果

每个任务对话框都有执行结果,用户可以拿到该结果进行进一步的操作和处理。Revit® API 提供了 TaskDialogResult 枚举来标识用户在任务对话框显示后的响应操作类

别。下面为该枚举的全部值。

- TaskDialogResult. None;
- TaskDialogResult. OK;
- TaskDialogResult. Cancel;
- TaskDialogResult. CommandLink1;
- TaskDialogResult. CommandLink2;
- TaskDialogResult. CommandLink3;
- TaskDialogResult. CommandLink4;
- TaskDialogResult. Retry;
- TaskDialogResult. Yes;
- TaskDialogResult. No;
- TaskDialogResult. Close。

第 11 章　Revit®结构建模

本章将介绍在 Revit®结构产品中特有的一些 API 特性：

① 结构模型元素：包括只在 Revit®结构产品中存在的特定元素及其属性，见 11.1 节。

② 分析模型：包括分析模型类、分析模型几何信息、分析模型参数信息、分析模型支撑信息、编辑分析模型和分析链接等，见 11.2 节。

11.1　结构模型元素

11.1.1　基本元素

1）区分结构柱，结构梁，结构支撑，独立基础

在 Revit® API 中结构柱、结构梁、结构支撑和独立基础都是 FamilyInstance 类，它们没有各自特定 API 类型。虽然它们都表现了 FamilyInstance 类型，但可以通过 StructuralType 属性来做区分，见代码片段 11-1。

代码片段 11-1：通过 StructuralType 区分结构柱、结构梁、结构支撑和独立基础

```
public void GetStructuralType(FamilyInstance familyInstance)
{
   string message = "";
   switch (familyInstance.StructuralType)
{
case StructuralType.Beam: //结构梁
        message = "FamilyInstance is a beam.";
break;
case StructuralType.Brace: //结构支撑
        message = "FamilyInstance is a brace.";
        break;
      case StructuralType.Column: //结构柱
        message = "FamilyInstance is a column.";
        break;
      case StructuralType.Footing: //独立地基
        message = "FamilyInstance is a footing.";
        break;
      default:
        message = "FamilyInstance is non-structural or unknown framing.";
        break;
   }
}
```

2）梁系统（BeamSystem）

梁系统，如图 11-1 所示。用户通过 BeamSystem 类获取和编辑梁系统。BeamSystem 类提供了一系列属性来得到和设置梁系统，如 BeamSystemType，BeamType，Direction，

Level 和 Profile 等。BeamSystem 类还提供一系列 Create 重载方法来创建梁系统。梁系统的类型表现为 BeamSystemType 类，可以通过 BeamSystem. BeamSystemType 属性来获取或修改当前梁系统的类型。

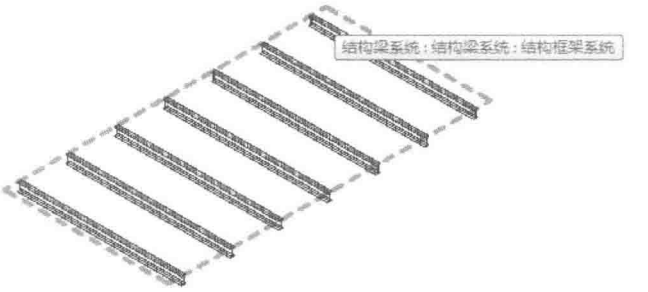


图 11-1　梁系统

3）结构桁架（Truss）

结构桁架，如图 11-2 所示。用户通过 Truss 类获取和编辑梁结构桁架，通过 Truss. Create 静态方法创建一个结构桁架。桁架的类型表现为 TrussType 类，可以通过 Truss. Truss 属性来获取或修改桁架类型。

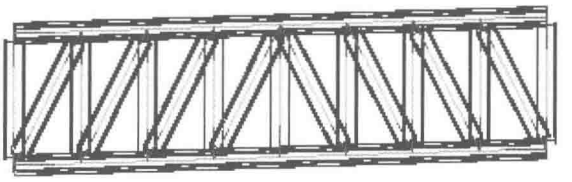


图 11-2　结构桁架

4）结构基础（ContFooting）

结构基础可分为独立基础，墙基础和基础底板。在 Revit® API 中，独立基础表现为 FaimilyInstance 类，墙基础表现为 ContFooting 类，基础底板表现为 Floor 类。

5）其他结构元素

在 Revit® API 中，一些元素既存在于结构模型中也存在于建筑模型中，如 Wall 类和 Floor 类。结构墙和建筑墙都表现为 Wall 类（关于 Wall 类的具体用法，请参照墙章节）。结构楼板和基础底板都是结构模型，它们和建筑楼板一样在 Revit® API 中都体现为 Floor 类（关于 Floor 类的具体用法，请参照地板章节）。

11.1.2　荷载

在 Revit® 中可以应用点、线和面荷载。通过绘制或使用主体构件（如梁、支撑、结构柱、结构楼板和结构墙）来应用荷载。在放置荷载前后，可以编辑荷载力（Forces）和弯矩参数（Moments）。可以修改荷载性质（Load Natures）和荷载工况（Load Cases）。也可以将荷载组合（Load Combinations）应用于模型。

1）荷载工况和荷载性质

如图 11-3 所示，在 Revit® API 中荷载工况表现为 LoadCase 类，可以通过 Autodesk. Revit. Creation. Document. NewLoadCase（string name，LoadNature nature，Category category）方法创建新的荷载工况。

荷载性质表现为 LoadNature 类，通过 Autodesk. Revit. Creation. Document. NewLoadNature（string name）方法创建新的荷载性质。

2）荷载组合

如图 11-4 所示，在 Revit® API 中，荷载组合表现为 LoadCombination 类，用户可以通过 Autodesk. Revit. Creation. Document. NewLoadCombination（string name，int typeInd，int stateInd，double [] factors，LoadCaseArray cases，LoadCombinationArray combinations，LoadUsageArray usages）方法创建荷载组合。



图 11-3　荷载工况设置对话框

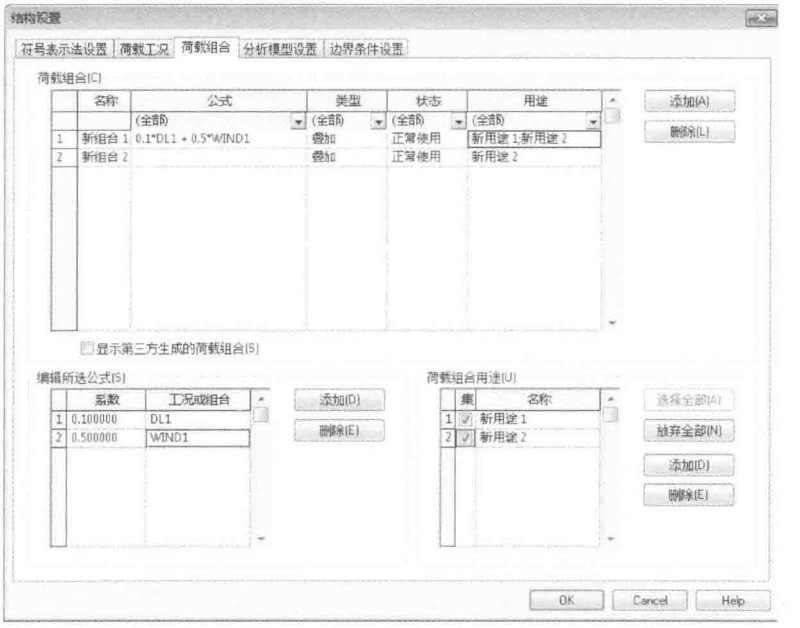


图 11-4　荷载组合设置对话框

荷载组合用途表现为 LoadUsage 类，通过 Autodesk. Revit. Creation. Document. NewLoadUsage(string name)创建。

3）荷载

在 Revit® 中可以应用点、线和面荷载。

(1) 点荷载

如图 11-5 所示，在 Revit® API 中点荷载表现为 PointLoad 类。Autodesk. Revit. Creation. Document 类提供了两个 NewPointLoad 重载方法来创建点荷载。

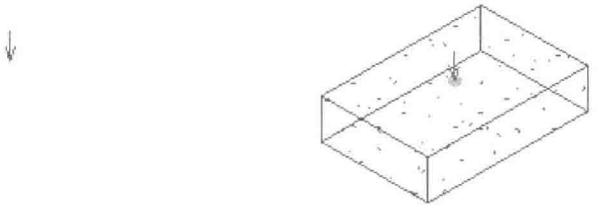


图 11-5　点荷载与主体点荷载

(2) 线荷载

如图 11-6 所示，在 Revit® API 中线荷载表现为 LineLoad 类。Autodesk. Revit. Creation. Document 类提供了三个 NewLineLoad 重载方法来创建线荷载。

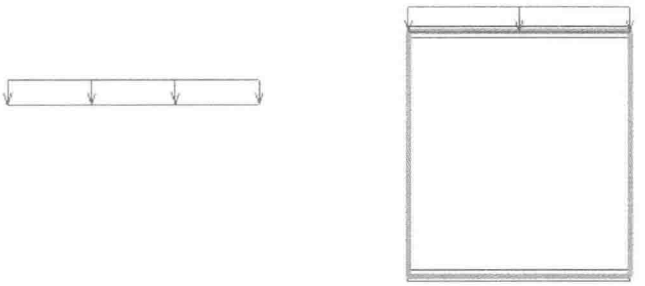


图 11-6　线荷载与主体线荷载

(3) 面荷载

如图 11-7 所示，在 Revit® API 中面荷载表现为 AreaLoad 类。Autodesk. Revit. Creation. Document 类提供了两个 NewAreaLoad 重载方法来创建面荷载。

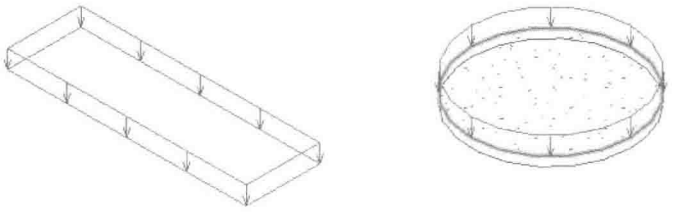


图 11-7　面荷载与主体面荷载

11.1.3　边界条件

边界条件是指根据其周围环境定义结构元素的支撑情况，也称支撑或约束。在 Revit® API 中表现为 BoundaryConditions 类。

1) 边界条件的分类和几何信息

边界条件可以分为点、线和面边界条件，在 Revit® API 中它们都对应 BoundaryConditions

类。那么在 Revit® API 中如何来区分呢？代码片段 11-2 展示了如何获取边界条件的类别与几何信息。

代码片段 11-2：获取边界条件的类别与几何信息

```
//BoundaryConditions BC
//通过 BOUNDARY_CONDITIONS_TYPE 参数获取此边界条件的类别
Parameter param =  BC.get_Parameter(BuiltInParameter.BOUNDARY_CONDITIONS_TYPE);
switch (param.AsInteger())
{
case 0: //点边界条件，几何信息表现为一个点
XYZ point =  BC.Point;
break;
case 1: //线边界条件，几何信息表现为一条线
Curve curve =  BC.get_Curve(0);
break;
case 2: //面边界条件，几何信息表现多条线的集合
CurveArray profile =  new CurveArray();
for (int i =  0; i <  BC.NumCurves; i+ + )
{
profile.Append(BC.get_Curve(i));
}
      break;
default:
break;
}
```

2）创建边界条件

Autodesk. Revit. Creation. Document 类提供了五个方法来创建边界条件。通过 NewPointBoundaryConditions 方法创建点边界条件；通过 NewLineBoundaryConditions 重载方法创建线边界条件；通过 NewAreaBoundaryConditions 重载方法创建面边界条件。代码片段 11-3 展示了如何在结构柱上创建线边界条件。

代码片段 11-3：在结构柱上创建线边界条件

```
//FamilyInstance beam
//Autodesk.Revit.Creation.Document creDocument
AnalyticalModel beamAM =  beam.GetAnalyticalModel();
Curve curve =  beamAM.GetCurve();
AnalyticalModelSelector selector =  new AnalyticalModelSelector(curve);
selector.CurveSelector =  AnalyticalCurveSelector.WholeCurve;
Reference wholeCurveRef =  beamAM.GetReference(selector);

//通过 NewLineBoundaryConditions 重载方法一创建线边界条件
BoundaryConditions createdBC1 =  creDocument.NewLineBoundaryConditions(beamAM,
TranslationRotationValue.Release, 0,
                                          TranslationRotationValue.Fixed, 0,
                                          TranslationRotationValue.Fixed, 0,
                                          TranslationRotationValue.Fixed, 0);
```

```
//通过 NewLineBoundaryConditions 重载方法二创建线边界条件
BoundaryConditions createdBC2 =  creDocument.NewLineBoundaryConditions(wholeCurveRef,
TranslationRotationValue.Release, 0,
TranslationRotationValue.Fixed, 0,
TranslationRotationValue.Fixed, 0,
TranslationRotationValue.Fixed, 0);
```

11.2 分析模型(AnalyticalModel)

11.2.1 概述

在 Revit®结构中,分析模型是对结构物理模型在工程学上的描述。

1) 有分析模型的元素

- 结构柱
- 结构框架(包括结构梁和结构支撑)
- 结构墙
- 结构楼板
- 结构基础(包括独立基础,条状基础和基础底板)

2) 获取分析模型

分析模型可以通过 Element. GetAnalyticalModel()方法得到。如果该元素不含有分析模型,Element. GetAnalyticalModel()将返回空。

需要注意的是,对于用户在程序里通过 API 的 Create 方法新创建的结构元素,它的分析模式是不可获取的,Element. GetAnalyticalModel()将返回空,直到 Regeneration 发生。为了避免这种情况发生,用户可以创建结构元素后调用 Document. Regenerate()方法,然后再调用 Element. GetAnalyticalModel()来获取分析模型,如代码片段 11-4 所示。

代码片段 11-4:获取分析模型

```
CurveArray profile =  new CurveArray();
// API 创建一个结构楼板
Floor struFloor =  RevitDoc.Create.NewFloor(profile, true);
// Regenerate revit document
RevitDoc.Regenerate();
//获取改楼板的分析模型
AnalyticalModel floorAM =  struFloor.GetAnalyticalModel();
```

3) 分析模型包含的信息

- 分析模型的位置信息(Analytical Location);
- 参数信息,包括延伸/对齐方法(AlignmentMethod)、投影(Projection)、分析为(AnalyzeAs)、近似曲线(Approximate)等;
- 支撑信息(AnalyticalModelSupport);
- 分析调整,包括手动调整(AlignmentMethod. AutoDetect)和自动调整(ManuallyAdjust)。

11.2.2　分析模型的几何信息

根据分析模型对应的元素的类型不同，分析模型的几何信息可以分别通过以下三个方法之一得到：GetPoint（）；GetCurve（）；GetCurves（AnalyticalCurveType curveType）。

需要注意的是，通过这几个方法得到线是没有对应的几何引用（Reference）的。因此不能基于这些线使用获取参照信息相关的 API 属性，如 Curve. Reference，Curve. EndPointReference。取而代之，Revit® 提供了 AnalyticalModelSelector 类获取对应的参照信息，详细介绍参见 11. 2. 3 节。

1）GetPoint

如果分析模型被表达为一个点（比如独立基础的分析模型），GetPoin 方法将返回这个点，否则 GetPoint 方法抛出 InapplicableDataException。可以通过 IsSinglePoint 方法判断该分析模型是否表达为一个点。

代码片段 11-5：获取独立基础的分析模型几何信息

```
//该元素是独立基地
FamilyInstance familyInst = element as FamilyInstance;
if (null != familyInst && familyInst.StructuralType == StructuralType.Footing)
{
AnalyticalModel model = familyInst.GetAnalyticalModel();
//独立地基的分析模型被表达为一个点
if (model.IsSinglePoint() == true)
{
XYZ analyticalLocationPoint = model.GetPoint();
}
}
```

2）GetCurve

如果分析模型被表达为一条线（如结构柱、结构梁），GetCurve 方法返回这条线，否则 GetCurve 方法抛出 InapplicableDataException。可以通过 IsSingleCurve 方法判断该分析模型是否表达为一条线，如代码片段 11-6 所示。

代码片段 11-6：获取结构柱的分析模型几何信息

```
//该元素为结构柱
FamilyInstance familyInst = element as FamilyInstance;
if (null != familyInst && familyInst.StructuralType == StructuralType.Column)
{
AnalyticalModel model = familyInst.GetAnalyticalModel();
//结构柱的分析结构被表达为一条线
if (model.IsSingleCurve() == true)
{
XYZ analyticalLocationPoint = model.GetCurve ();
}
}
```

3) GetCurves(AnalyticalCurveType curveType)

如果分析模型被表达为多条线(如结构墙、结构楼板),使用 GetCurves 方法来获取这些线。

事实上,所有的分析模型都可以通过 GetCures 方法来获取几何信息。当分析模型被表达为一个点时,GetCurves 返回的线段列表里只包含一条线,并且这条线的长度几乎为零;当分析模型被表达为一条线时,GetCurves 返回的线段列表里只包含一条线。

GetCurves 方法需要传递一个 AnalyticalCurveType 枚举值。在 GetCurves 里用到的枚举成员有:

- BaseCurve:默认分析模型线。BaseCurve 只适用于线形元素(比如结构柱,结构梁和结构支撑),不适用于结构墙、结构楼板等。
- RawCurves:生成的基础分析模型线。
- ActiveCurves:显示在屏幕上的分析模型线(不包括分析链接)。
- ApproximatedCurves:近似曲线。

AnalyticalCurveType 枚举还包含 RigidLinkHead,RigidLinkTail 和 AllRigidLinks 这三个与分析链接相关的成员,详细使用方法请参考本章的分析链接一节。

代码片段 11-7:获取结构墙的分析模型几何信息

```
Wall wall = null;
//获取结构墙的分析模型
AnalyticalModel amWall = wall.GetAnalyticalModel() as AnalyticalModel;
if (null == amWall)
{
//建筑墙没有分析模型
return;
}
//获取分析模型曲线
int modelCurveNum = amWall.GetCurves(AnalyticalCurveType.ActiveCurves).Count;
```

11.2.3 分析模型曲线的几何引用(Reference)

分析模型曲线的几何引用可以用作参数来创建某些结构元素(如 Creation. Document. NewPointBoundaryConditions,Creation. Document. NewPointLoad 等)。

用户通过 AnalyticalModel. GetReference(AnalyticalModelSelector selector)方法获取分析模型曲线的几何引用,通过 selector 参数指定是获取整条曲线的几何引用,曲线起点的参照,还是曲线终点的几何引用。

在调用 AnalyticalModel. GetReference 前可以通过 AnalyticalModel. IsValidSelector (AnalyticalModelSelector selector)来检验证查此 selector 是否可用于该分析模型。

下面将通过具体例子进行介绍。

① 分析模型为一个点的几何引用获取。代码片段 11-8 展示了如何获取独立基础的分析模型点的参照。

代码片段 11-8：获取独立基础的分析模型点的参照

```
//element: 一个独立基础
//获取独立基础的分析模型
AnalyticalModel am =  element.GetAnalyticalModel();
AnalyticalModelSelector amSelector =  new AnalyticalModelSelector();
//获取独立基础的分析模型点的几何引用
Reference refer =  am.GetReference(amSelector);
```

② 分析模型为一条线的几何引用获取。代码片段 11-9 展示了如何获取结构梁的分析模型曲线的几何引用。

代码片段 11-9：获取结构梁的分析模型曲线的几何引用

```
//element:一个结构梁
//获取结构梁的分析模型
AnalyticalModel am =  element.GetAnalyticalModel();

//获取结构梁的分析模型曲线的几何引用
AnalyticalModelSelector  amSelector  =  newAnalyticalModelSelector ( AnalyticalCurveSelector.
WholeCurve);
Reference wholeCurveRefer =  am.GetReference(amSelector);

//获取结构梁的分析模型曲线的起点的几何引用
amSelector =  new AnalyticalModelSelector(AnalyticalCurveSelector.StartPoint);
Reference curveStartPointRefer =  am.GetReference(amSelector);

//获取结构梁的分析模型曲线的终点的几何引用
amSelector =  new AnalyticalModelSelector(AnalyticalCurveSelector.EndPoint);
Reference curveEndPointRefer =  am.GetReference(amSelector);
```

③ 分析模型为多条线的几何引用获取。代码片段 11-10 展示了获取结构墙的分析模型曲线的参照。

代码片段 11-10：获取结构墙的分析模型曲线的参照

```
//element:一个结构墙
//获取结构墙的分析模型
AnalyticalModel am =  element.GetAnalyticalModel();
//获取结构墙分析模型的曲线列表
IList< Curve>  amCurveList =  am.GetCurves(AnalyticalCurveType.ActiveCurves);

for (int curveIndex =  0; curveIndex <  amCurveList.Count; curveIndex+ + )
{
//获取结构墙的指定的分析模型曲线的几何引用
AnalyticalModelSelector  amSelector  =  new  AnalyticalModelSelector  ( amCurveList
[curveIndex], AnalyticalCurveSelector.WholeCurve);
Reference wholeCurveRefer =  am.GetReference(amSelector);

//获取结构墙的指定的分析模型曲线的起点的几何引用
```

```
amSelector =  new AnalyticalModelSelector(amCurveList[curveIndex], AnalyticalCurveSelector.
StartPoint);
Reference curveStartPointRefer =  am.GetReference(amSelector);

//获取结构墙的指定的分析模型曲线的终点的几何引用
amSelector =  new AnalyticalModelSelector(amCurveList[curveIndex], AnalyticalCurveSelector.
EndPoint);
Reference curveEndPointRefer =  am.GetReference(amSelector);
}
```

11.2.4 分析模型参数相关的 API

1）自动调整(Alignment)

分析模型有一些对齐/延伸方法相关的属性。当用户将这些属性设置为自动检测，分析模型就会在需要的时候自动进行微调。例如，在墙与墙相连时，或者梁和邻近的梁之间，分析模型会自动伸出或缩进等以保证分析模型间的连通性，见图 11-8。

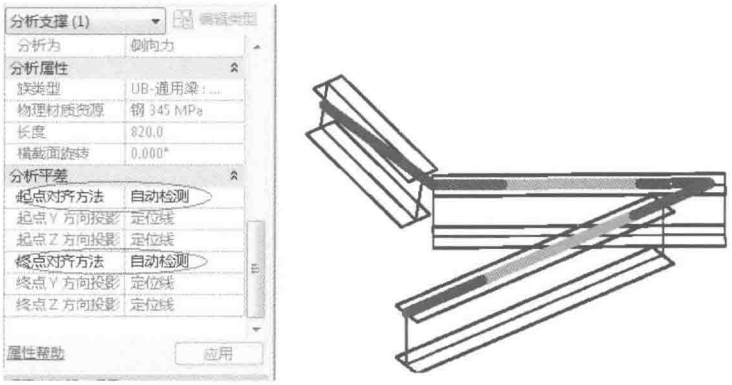


图 11-8 分析模型对齐/延伸方式的自动调整

用户可以通过 AnalyticalModel. SetAlignmentMethod 方法将对应的对齐/延伸方设置为 AnalyticalAlignmentMethod. AutoDetect。

2）投影(Projection)

用户可以通过 AnalyticalModel. GetAnalyticalProjectionType 和 AnalyticalModel. SetAnalyticalProjectionType 方法获取或者设置分析模型的投影相关的参数。

需要注意的是，在设置投影参数之前，必须保证对齐/延伸方法参数的值为投影，否则 AnalyticalModel. SetAnalyticalProjectionType 将抛出异常。用户可以通过 AnalyticalModel AnalyticalModel. SetAlignmentMethod 方法设置对齐/延伸方法，需要将对齐方式设置为 AnalyticalAlignmentMethod. Projection，见图 11-9。



图 11-9 分析模型的对齐/延伸方式设置为投影

代码片段 11-11 展示了设置结构柱分析模型底部 Z 方向投影为“柱底部”。

代码片段 11-11：设置结构柱分析模型底部 Z 方向投影为“柱底部”

```
am. SetAlignmentMethod (AnalyticalElementSelector. StartOrBase, AnalyticalDirection. Z,
    AnalyticalAlignmentMethod.Projection);
am.SetAnalyticalProjectionType (AnalyticalElementSelector.StartOrBase, AnalyticalDirection.Z,
    AnalyticalProjectionType.Bottom);
```

• 投影到基准面

这里将专门介绍如何投影到基准面，因为不能通过 AnalyticalModel. SetAnalyticalProjectionType来实现。用户需要通过 AnalyticalModel. GetAnalyticalProjectionDatumPlane 方法得到投影到的基准面；通过 AnalyticalModel. SetAnalyticalProjectionDatumPlane 方法来设置投影到某个基准面。同样地，在设置投影到基准面之前，用户必须确认对齐/延伸方法参数的值为投影。如图 11-10 所示的是将分析模型的对齐/延伸方式设置为投影到某个基准面（标高 1），见代码示例 11-12。

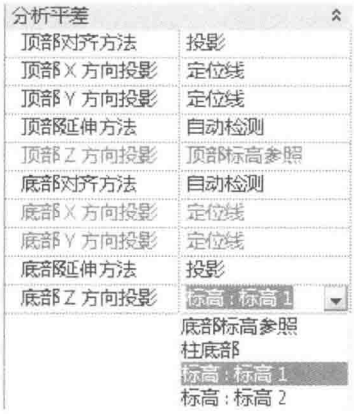


图 11-10　分析模型的对齐/延伸方式设置为投影到某个基准面（标高 1）

代码片段 11-12：设置结构柱分析模型底部 Z 方向投影为“标高 1”

```
am. SetAlignmentMethod (AnalyticalElementSelector. StartOrBase, AnalyticalDirection. Z,
    AnalyticalAlignmentMethod.Projection);
am.     SetAnalyticalProjectionDatumPlane     (     AnalyticalElementSelector.     StartOrBase,
    AnalyticalDirection. Z, level1Id);
```

3）近似曲线（Approximate）

当分析模型是一个曲线而不是直线的时候，近似曲线将会作为一个可选参数出现在属性对话框里见图 11-11。

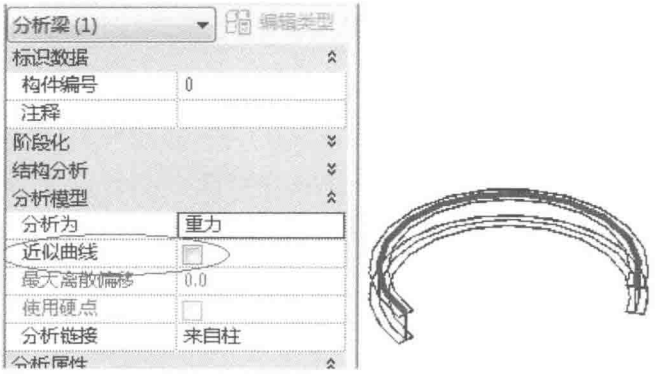


图 11-11　分析模型曲线

用户可以通过 AnalyticalModel. CanApproximate 方法判断该分析模型是否可以近似。

当 AnalyticalModel. CanApproximate 返回 true，通过 AnalyticalModel. Approximate (bool)方法设置为 true 或 false 来转换近似曲线和曲线。

当设置 AnalyticalModel. Approximate 为 true 时，通过 Analytical Model. SetApproximation Deviation(Double)来设置最大离散偏移值，通过 SetUsesHardPoints (bool)来设置或取消使用硬点。如图 11-12 所示的是近似分析模型曲线。代码片段 11-13 展示的是如何设置结构梁分析模型的近似曲线。

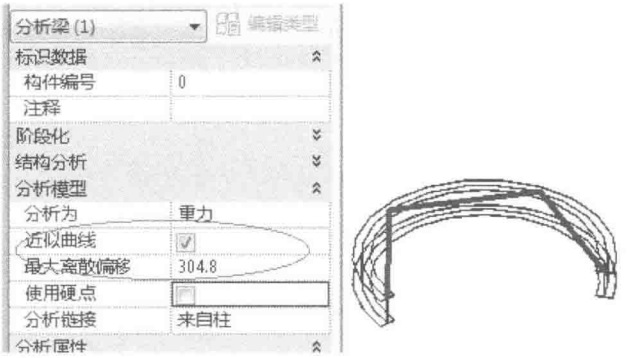


图 11-12 近似分析模型曲线

代码片段 11-13：设置结构梁分析模型的近似曲线

```
if (am.CanApproximate())
{
   am.Approximate(true);
   am.SetApproximationDeviation(1.5);
   am.SetUsesHardPoints(false);
}
```

4) 分析为(AnalyzeAs)

用户可以通过 AnalyticalModel. GetAnalyzeAs 和 AnalyticalModel. SetAnalyzeAs 方法来获取或设置分析为参数的值。该参数值对应的 API 类型为 AnalyzeAs 枚举，需要注意的是，并不是所有的 AnalyzeAs 成员值都可用于所有的分析模型。在调用 AnalyticalModel. SetAnalyzeAs 之前，用户可以通过 AnalyticalModel. IsAnalyzeAsValid 方法判断指定的 AnalyzeAs 成员值是否可用于该分析模型。如图 11-13所示的是“分析为”属性的下拉菜单。代码片段 11-14 展示的是如何设置分析模型的“分析为”属性。

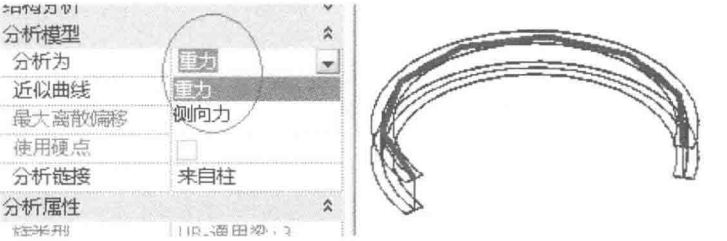


图 11-13 “分析为”属性

代码片段 11-14：设置分析模型的“分析为”属性

```
if(am.IsAnalyzeAsValid(AnalyzeAs.Gravity))
{
   am.SetAnalyzeAs(AnalyzeAs.Gravity);
}
```

11.2.5　支撑信息(AnalyticalModelSupport)

分析模型提供了 AnalyticalModel. IsElementFullySupported 方法来判断该分析模型是否被完全支撑。更多的支撑信息可以通过 AnalyticalModel. GetAnalyticalModelSupports 方法获取,该方法将返回一个 AnalyticalModelSupport 类型的列表,每个 AnalyticalModelSupport 成员表示一个支撑。通过 AnalyticalModelSupport 可以获取具体的点、线、面等支撑信息。

下面以具体的例子来阐述不同的情况下如何使用 AnalyticalModelSupport。

1) 楼板和结构梁

以拾取支座的方式,选择三个分析梁作为支座创建楼板,见图 11-14。

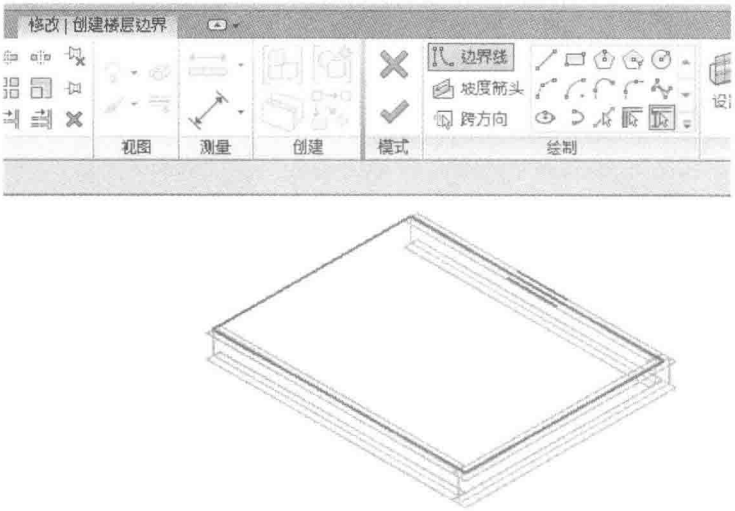


图 11-14　拾取三根结构梁为支座创建结构楼板

通过 API 来分析该楼板的支撑信息,见代码片段 11-15。

代码片段 11-15:楼板的支撑信息

```
public void GetSupportInfo_Floor(Floor floor)
{
   // floor :楼板
//获取楼板的分析模型
   AnalyticalModel amFloor =  floor.GetAnalyticalModel();
   //在这里 IsElementFullySupported 方法将返回 true,这表明楼板被完全支撑
   bool fullySuported =  amFloor.IsElementFullySupported();

   //获取具体的支撑信息,将返回含有三个 support 成员的列表。每一个 support 都对应一根结构梁
     对地板提供的支撑
IList< AnalyticalModelSupport>  supportList =  amFloor.GetAnalyticalModelSupports();
foreach (AnalyticalModelSupport support in supportList)
   {
      //返回当前提供支撑的分析梁的 id
      ElementId id =  support.GetSupportingElement();
```

```
        //返回 CurveSupport,表明该支撑是一个线性支撑
        AnalyticalSupportType supportType =  support.GetSupportType();

        //根据支撑力类型的不同,可以分别通过 GetPoint, GetCurve, GetFace 这三种方法之一获取支
          撑力的具体几何位置
        switch (supportType)
        {
           case AnalyticalSupportType.PointSupport:
              XYZ point =  support.GetPoint();
              break;
              case AnalyticalSupportType.CurveSupport:
              Curve curve =  support.GetCurve();
              break;
           case AnalyticalSupportType.SurfaceSupport:
              Face face =  support.GetFace();
              break;
           case AnalyticalSupportType.UnknownSupport:
              break;
        }

        //通过 GetPriority 方法可以获取当前支撑力的优先权。当一个物体被多项支撑时,该属性可
          对提供的支撑进行排序,从而可选取最优的支撑
        AnalyticalSupportPriority supportPriority =  support.GetPriority();
    }
}
```

通过这段代码可以看出,在这个示例中有三条梁支撑着楼板。

2) 楼板和墙

以拾取墙的方式画一个楼板,如图 11-15 所示。

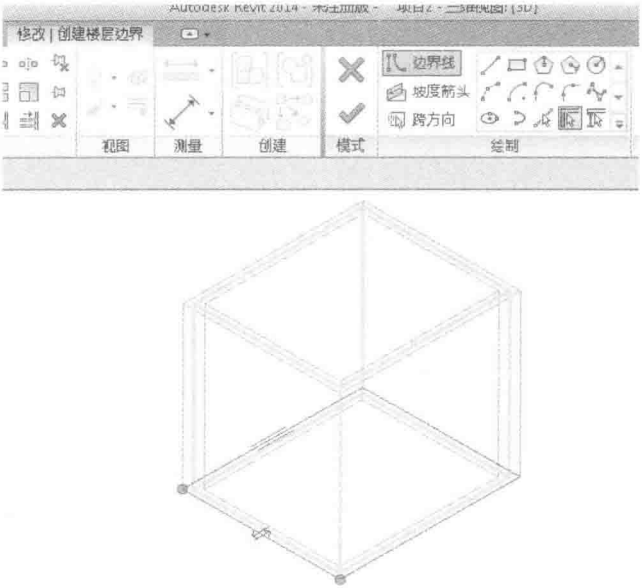


图 11-15　拾取四面结构墙创建结构楼板

通过 API 来分析该楼板和墙之间的支撑信息，见代码片段 11-16 和代码片段 11-17。

代码片段 11-16：楼板的支撑信息

```
//楼板的支撑信息
public void GetSupportInfo_Floor(Floor floor)
{
   // floor :楼板
//获取楼板的分析模型
   AnalyticalModel amFloor =  floor.GetAnalyticalModel();
   //在这里 IsElementFullySupported 方法将返回 false，这表明楼板未被完全支撑
   bool fullySuported =  amFloor.IsElementFullySupported();

   //获取具体的支撑信息，将返回一个空列表。表示该楼板没有任何支撑物
IList< AnalyticalModelSupport>  supportList =  amFloor.GetAnalyticalModelSupports();
If(supportList.Count = =  0) string message = "楼板没有支撑";
}
```

代码片段 11-17：墙的支撑信息

```
//该实例中墙的支撑信息
public void GetSupportInfo_Walls(Wall wall)
{
   //wall : 四面墙中的任何一面墙
//获取墙的分析模型
   AnalyticalModel amWall =  wall.GetAnalyticalModel();
   //在这里 IsElementFullySupported 方法将返回 true，这表明墙被完全支撑
   bool fullySuported =  amWall.IsElementFullySupported();

   //获取具体的支撑信息，这里返回只含有一个 support 成员的列表。这个 support 对应墙下面的
     楼板
IList< AnalyticalModelSupport>  supportList =  amWall.GetAnalyticalModelSupports();
foreach (AnalyticalModelSupport support in supportList)
   {
      //返回当前提供支撑的楼板的分析模型的 id
      ElementId id =  support.GetSupportingElement();
      //返回 CurveSupport，表明该支撑是一个线性支撑
      AnalyticalSupportType supportType =  support.GetSupportType();

      //根据支撑力类型的不同，可以分别通过 GetPoint，GetCurve，GetFace 这三种方法之一获取支
        撑力的具体几何位置
      switch (supportType)
      {
         case AnalyticalSupportType.PointSupport:
            XYZ point =  support.GetPoint();
            break;
            case AnalyticalSupportType.CurveSupport:
            Curve curve =  support.GetCurve();
            break;
         case AnalyticalSupportType.SurfaceSupport:
            Face face =  support.GetFace();
```

```
            break;
        case AnalyticalSupportType.UnknownSupport:
            break;
    }

    AnalyticalSupportPriority supportPriority = support.GetPriority();
  }
}
```

通过这两段代码可以看出，在这个示例中，墙并不对它下面的楼板提供支撑，而是楼板支撑着它上面的四面墙。

3）结构柱和结构梁

通过 API 来分析该结构梁（图 11-16）的支撑信息详见代码片段 11-18。

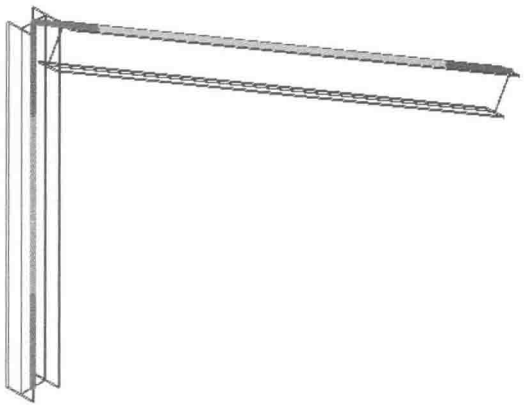


图 11-16　相连接的结构柱和结构梁

代码片段 11-18：结构梁的支撑信息

```
//该实例中梁的支撑信息
public void GetSupportInfo_Beam(FamilyInstancebeam)
{
  //beam: 结构梁
//获取梁的分析模型
  AnalyticalModel amBeam = beam.GetAnalyticalModel();
  //在这里 IsElementFullySupported 方法将返回 false，这表明梁未被完全支撑
  bool fullySuported = amBeam.IsElementFullySupported();

  //获取具体的支撑信息，这里返回只含有一个 support 成员的列表。这个 support 对应该示例中的
    结构柱
IList< AnalyticalModelSupport> supportList = amBeam.GetAnalyticalModelSupports();
foreach (AnalyticalModelSupport support in supportList)
  {
    //返回提供该支撑的结构柱的分析模型的 id
    ElementId id = support.GetSupportingElement();
    //返回 PointSupport，表明该支撑是一个点支撑
    AnalyticalSupportType supportType = support.GetSupportType();
```

```
        //根据支撑力类型的不同,可以分别通过 GetPoint,GetCurve,GetFace 这三种方法之一获取支
          撑力的具体几何位置
        switch (supportType)
        {
           case AnalyticalSupportType.PointSupport:
              XYZ point =  support.GetPoint();
              break;
              case AnalyticalSupportType.CurveSupport:
              Curve curve =  support.GetCurve();
              break;
           case AnalyticalSupportType.SurfaceSupport:
              Face face =  support.GetFace();
              break;
           case AnalyticalSupportType.UnknownSupport:
              break;
        }

        AnalyticalSupportPriority supportPriority =  support.GetPriority();
    }
}
```

通过这段代码可以看出,该结构梁的一端被一根结构柱支撑着;但是该梁没有被完全支撑,因为它只有一端被柱子支撑,另一端还悬空着,是不稳定支撑。

11.2.6　编辑分析模型

分析模型与物理模型的位置具有一致性,它通常有一个默认位置跟随着物理模型。在某些情况下分析模型会自动微调(请参考分析模型参数相关的 API 章节的自动调整章节)。也可以通过设置投影方式来修改分析模型的位置(这部分在分析模型参数相关的 API 章节的投影一节已经讲解过)。用户还可以通过设置偏移量,或者手动调整来编辑分析模型,本章我们将着重对这两部分进行讲解。

1) 设置线性分析模型的端点偏移量

用户可以通过 AnalyticalModel. SetOffset (AnalyticalElementSelector selector, XYZ offset)方法来设置线性分析模型端点的偏移量,通过 selector 参数来指定被设置偏移量的端点,通过 offet 参数来指定偏移量。通过 AnalyticalModel. GetOffset (AnalyticalElement Selector selector)获取分析模型端点的偏移量。

注意:SetOffset 是专门针对线性分析模型的,如结构梁柱等;结构墙,楼板等的分析模型不支持设置偏移量。

在 Revit® 界面调整结构柱分析模型的顶端向上(Z 方向)偏移,见图 11-17 和代码片段 11-19。





图 11-17　编辑分析模型的端点

代码片段 11-19:调整结构柱分析模型的顶端向上(Z 方向)偏移 5

```
// columnAnalytical 结构柱的分析模型
AnalyticalElementSelector elemSelector =  AnalyticalElementSelector.EndOrTop;
XYZ offset =  new XYZ(0, 0, 5);
columnAnalytical.SetOffset(elemSelector, offset);

//返回 XYZ(0, 0, 5);
XYZregetOffset =  columnAnalytical.GetOffset(elemSelector);
```

2) 手动调整

分析模型还可以通过手动调整来与其他分析模型相连接。用户首先通过 AnalyticalModel. SupportsManualAdjustment 方法判断该分析模型是否允许被手动调整。如果允许,用户可以通过 AnalyticalModel. ManuallyAdjust 方法来手动调整分析模型(图 11-18 和图 11-19)。用户还可以通过 AnalyticalModel. ResetManualAdjustment 方法将分析模型设置回默认位置。

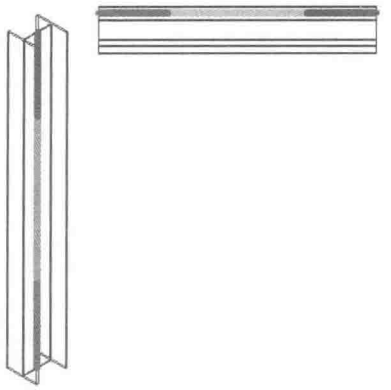


图 11-18 没有互相连接的分析柱和分析梁

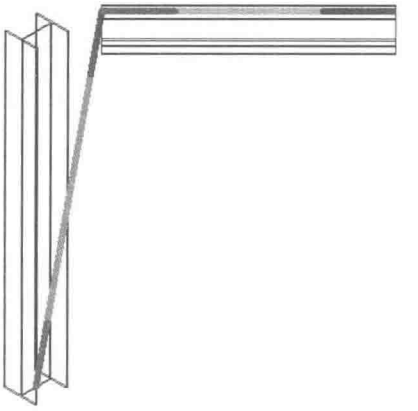


图 11-19 手动编辑分析柱的顶点,使其与分析梁相连接

代码片段 11-20:通过手动调整使结构柱的分析模型与某结构梁的分析模型相连接

```
// columnAnalytical:分析柱
// beamAnalytical 分析梁
//判断该分析模型支持手动调整
if(columnAnalytical.SupportsManualAdjustment())
{
Reference  sourceRef  =  columnAnalytical.  GetReference  ( newAnalyticalModelSelector
    (AnalyticalCurveSelector.EndPoint));
   Reference  targetRef  =  beamAnalytical. GetReference ( new  AnalyticalModelSelector
   (AnalyticalCurveSelector.EndPoint));
  //手动调整分析柱的终点到分析梁的终点
columnAnalytical.ManuallyAdjust(sourceRef, targetRef);
  bool isManuallyAdjusted = columnAnalytical.IsManuallyAdjusted());//true
```

```
   //将分析模型设置回默认位置
columnAnalytical.ResetManualAdjustment();
   isManuallyAdjusted = columnAnalytical.IsManuallyAdjusted()); //true
}
```

11.2.7　分析链接(AnalyticalLink)

分析链接是两个分析模型间的一个链接，它表现为一个单独的元素 AnalyticalLink，可以通过遍历 Document 得到所有的分析链接。在分析柱和分析梁之间可以根据一定的规则自动的生成分析链接(图 11-20)。也可以在任意分析模型间手动创建分析链接(图 11-21)。

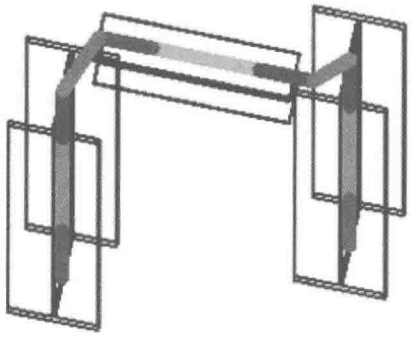


图 11-20　分析柱和临近的分析梁间自动生成分析链接

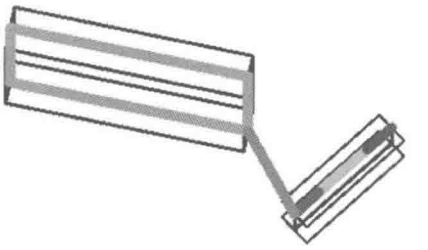


图 11-21　在分析墙和分析梁之间手动创建分析链接

1) 读取分析链接元素

可以通过遍历 Revit® 文档得到所有的分析链接元素，见代码片段 11-21。

代码片段 11-21：读取 Revit® 文档对象里所有的分析链接，包括自动创建的和手动创建的

```
public void ReadAnalyticalLinks(Document document)
{
//过滤出所有的分析链接元素
    FilteredElementCollector collectorAnalyticalLinks = new FilteredElementCollector
    (document);
   collectorAnalyticalLinks.OfClass(typeof(AnalyticalLink));

IEnumerable< AnalyticalLink> alinks = collectorAnalyticalLinks.
ToElements().Cast< AnalyticalLink> ();
  int nAutoGeneratedLinks = 0;
  int nManualLinks = 0;
  foreach (AnalyticalLink alink in alinks)
  {
//统计 Revit 自动创建的分析链接
      if (alink.IsAutoGenerated() = = true)
nAutoGeneratedLinks+ + ;
      else //统计手动创建的分析链接
nManualLinks+ + ;
```

```
}
string msg = "Auto-generated AnalyticalLinks:" + nAutoGeneratedLinks;
msg + = "\nManually created AnalyticalLinks:" + nManualLinks;
   TaskDialog.Show("AnalyticalLinks", msg);
}
```

2）如何通过程序创建分析链接

用户可以通过静态方法 AnalyticalLink. Create 来创建一个分析链接。需要注意的是，分析链接是链接两个分析模型，但 Create 方法并不是通过指定两个分析模型来创建，而是指定两个分析节点（在 API 中表现为 Hub 类）来创建。那么这里就出现一个问题，如何建立起分析模型和分析节点之间的对应关系，从而找到需要创建分析链接的分析节点，详见代码片段 11-22。

代码片段 11-22：选择两个族实例创建分析链接

```
public void CreateLink(Document doc, FamilyInstance fi1, FamilyInstance fi2)
{
//得到该 Document 里所有的分析节点
   FilteredElementCollector hubCollector = new FilteredElementCollector(doc);
   hubCollector.OfClass(typeof(Hub));
   ICollection< Element> allHubs = hubCollector.ToElements();
   //得到第一个 AnalyticalLinkType
   FilteredElementCollector linktypeCollector = new FilteredElementCollector(doc);
   linktypeCollector.OfClass(typeof(AnalyticalLinkType));
   ElementId firstLinkType = linktypeCollector.ToElementIds().First();

   //得到指定的族实例的分析节点
   ElementId startHubId = GetHub(fi1.GetAnalyticalModel().Id, allHubs);
   ElementId endHubId = GetHub(fi2.GetAnalyticalModel().Id, allHubs);

   Transaction tran = new Transaction(doc, "Create Link");
tran.Start();
   //通过指定两个分析节点来创建分析链接
   AnalyticalLink createdLink = AnalyticalLink.Create(doc, firstLinkType, startHubId,
   endHubId);
   tran.Commit();
}

/// < summary>
///获取指定的分析模型的第一个分析节点
/// < /summary>
/// < param name= "hostId"> 分析模型< /param>
/// < param name= "allHubs"> 分析节点集合< /param>
/// < returns> 返回指定的分析模型的第一个分析节点< /returns>
private ElementId GetHub(ElementId hostId, ICollection< Element> allHubs)
{
foreach (Element ehub in allHubs)
   {
Hub hub = ehub as Hub;
      ConnectorManager manager = hub.GetHubConnectorManager();
```

```
        ConnectorSet connectors =  manager.Connectors;
        foreach (Connector connector in connectors)
        {
ConnectorSet refConnectors =  connector.AllRefs;
foreach (Connector refConnector in refConnectors)
          {
if (refConnector.Owner.Id = =  hostId)
             {
return hub.Id;
             }
          }
}
}
return ElementId.InvalidElementId;
}
```

第 12 章　材料(Material)

12.1　材料简介

材料控制模型图元在视图和渲染图像中的显示方式。使用材料可以指定以下信息：

- 图形(仅限于 Revit®)：这些特性控制材料在未渲染视图中的外观。
- 外观：这些特性控制材料在渲染视图、真实视图或光线追踪视图中的显示方式。
- 物理：这些特性用于结构分析。
- 热度(仅限于 Revit®)：这些特性用于能量分析。

在 Revit®用户界面上，要使用材料，请单击“管理”选项卡➤“设置”面板“材料”。

在 Revit® API 中，使用 Material 类来代表材料，Material 类在 Autodesk. Revit. DB 命名空间中。上面所列的材料的四方面的信息都能从以下 API 类获取：

- 图形：Material 类；
- 外观：AppearanceAssetElement 类；
- 物理：PropertySetElement 类；
- 热度：PropertySetElement 类。

12.2　材料的标识

图 12-1 示例了“标识”页的信息，通过 Material 类的属性和参数来访问或者修改这些信息：

- 名字：Material. Name 属性；
- 类别：Material. MaterialClass 属性；
- 其他大部分属性可以在 Material 类的参数列表(Parameters)里面找到。

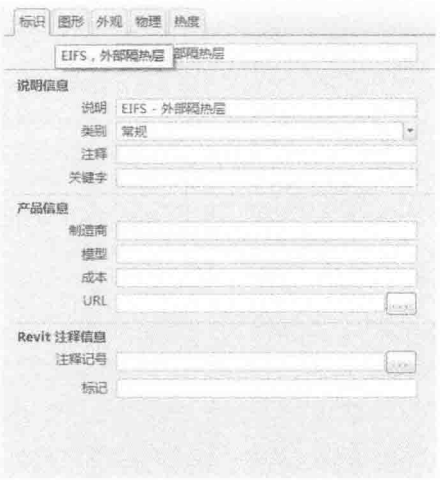


图 12-1　材料标识

12.3　材料的图形信息

图 12-2 示例了“图形”页的信息，通过 Material 类的属性来访问或者修改这些信息。

1) 着色

- 使用渲染外观：Material. UseRenderAppearanceForShading 属性
- 颜色：Material. Color 属性

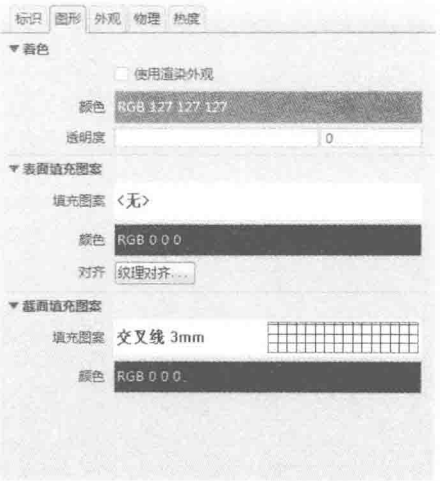


图 12-2　材料图形

- 透明度：Material. Transparency 属性

2）表面填充图案

- 填充图案：Material. SurfacePatternId 属性
- 颜色：Material. SurfacePatternColor 属性

3）截面填充图案

- 填充图案：Material. CutPatternId 属性
- 颜色：Material. CutPatternColor 属性

12.4　材料的外观信息

12.4.1　外观元素

图 12-3 示例了“外观”页的信息，这些信息在 Revit® 中被封装成了文档中的一个元素（在 API 中，对应了类 AppearanceAssetElement），材料通过连接此类元素来定义材料的外观。此类元素可以被文档中的所有材料访问。

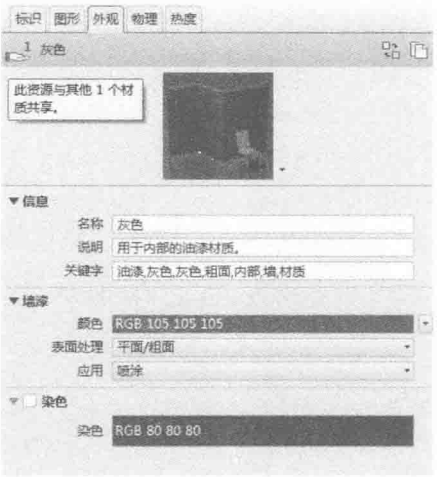


图 12-3　材料外观

代码片段 12-1 示例如何从材料（Material）实例中读写外观元素（AppearanceAssetElement）实例。

代码片段 12-1：读写外观元素

```
public void GetAndSetMaterialAppearance(Document doc, Material mat, Asset anotherAsset)
{
    ElementId assetElementId = mat.AppearanceAssetId;
    if (assetElementId ! = ElementId.InvalidElementId)
    {
        //获取材料的外观元素(AppearanceAssetElement)实例
        AppearanceAssetElement appearanceElement =
            doc.GetElement(assetElementId) as AppearanceAssetElement;

        //读写外观的属性集合(Asset)
        Asset currentAsset = appearanceElement.GetRenderingAsset();
        appearanceElement.SetRenderingAsset(anotherAsset);
    }

    //设置材料的外观元素。
    FilteredElementCollector cotr = new FilteredElementCollector(doc);
    IEnumerable< AppearanceAssetElement> allAppearanceElements =
        cotr.OfClass(typeof(AppearanceAssetElement)).Cast< AppearanceAssetElement> ();
    mat.AppearanceAssetId = allAppearanceElements.First< AppearanceAssetElement> ().Id;
}
```

注意：在 Revit® 中，AppearanceAssetElement 类代表的是文档中的外观元素，也只有已经从外观库中加载到了文档中的外观元素，才可以被 API 直接访问，其他在外观库中的外

观需要先加载到文档中(图 12-4)。代码片段 12-2 示例了如何把外观库的元素加载到文档。

代码片段 12-2:加载外观元素到文档

```
public void LoadAndGetAllAppearanceAssets(
      Autodesk. Revit. ApplicationServices.
Application revitApp)
{
     AssetSet theAssetSet =  revitApp. get_
     Assets(AssetType.Appearance);
    foreach (Asset theAsset in theAssetSet)
    {
        //可以在这里读取用 API 加载的外观元
素了。
    }
}
```



图 12-4　文档中和未加载的外观元素

12.4.2　访问外观属性集合

外观的详细信息被封装成了一个外观属性的集合(Asset 类,类图如图 12-5 所示),该集合的实例可以通过访问该元素的 Get 方法来获取。这部分的信息比较特殊,该信息和 Autodesk® 公司的材料库是紧密关联的,材料的外观属性随着材料会有较大的不同,而且会非常容易受到材料库的更新的影响。所以这部分的 API 也相对比较弱,API 中仅仅提供了读取该信息的方式,而且读取的方式也不是常见的通过属性来访问。

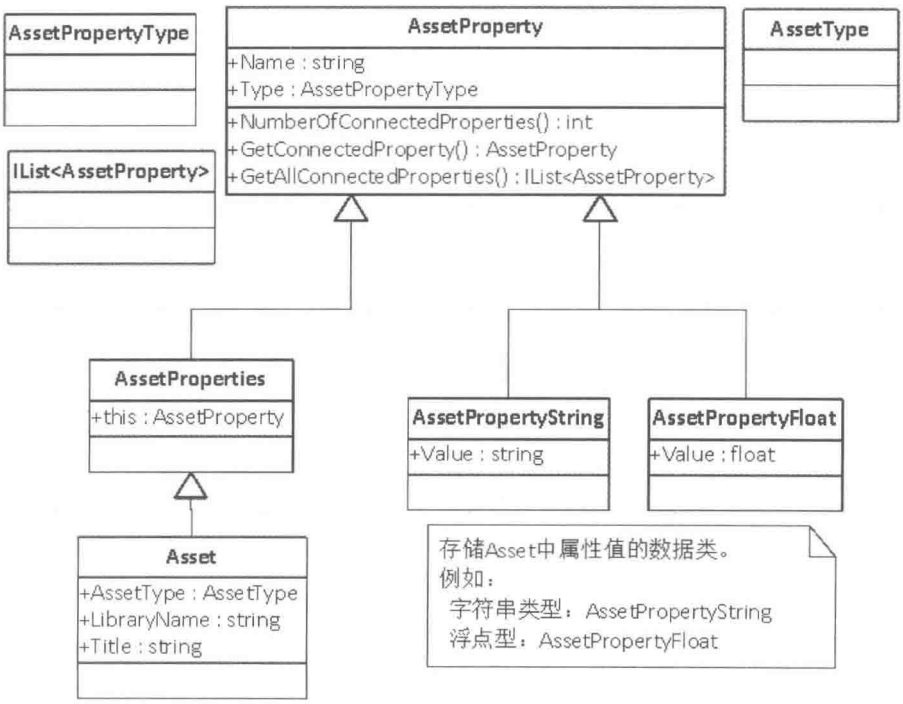


图 12-5　外观属性集合的类图

Asset 类实际上是一个储存外观属性的树。它的子节点可以是一个具体的叶节点(储存属性的名称和值),也可以是一个子树(可以得到另一个 Asset 类)。Asset 类可以访问的属性类型见表 12-1。

表 12-1　　Asset 类的属性类型

AssetProperty 的类型	使用方法
APT_Boolean	该 AssetProperty 是 AssetPropertyBoolean 类型,可以得到一个 bool 值
APT_Enum	该 AssetProperty 是 AssetPropertyEnum 类型,可以得到一个 int 值,该 int 指对应了内部使用的一个 Enum 的值
APT_Integer	该 AssetProperty 是 AssetPropertyInteger 类型,可以得到一个 int 值
APT_Int64	该 AssetProperty 是 AssetPropertyInt64 类型,可以得到一个 Int64 值
APT_UInt64	该 AssetProperty 是 AssetPropertyUInt64 类型,可以得到一个 UInt64 值
APT_Float	该 AssetProperty 是 AssetPropertyFloat 类型,可以得到一个 float 值
APT_Double	该 AssetProperty 是 AssetPropertyDouble 类型,可以得到一个 double 值
APT_String	该 AssetProperty 是 AssetPropertyString 类型,可以得到一个 string 值
APT_Time	该 AssetProperty 是 AssetPropertyTime 类型,可以得到一个 DateTime 值
APT_Distance	该 AssetProperty 是 AssetPropertyDistance 类型,可以得到一个长度的值,包括长度的值和单位
APT_DoubleArray2d	该 AssetProperty 是 AssetPropertyDoubleArray2d 类型,可以得到一个由 2 个 double 值组成的集合,该集合往往用来表示一个(U, V)值
APT_DoubleArray3d	该 AssetProperty 是 AssetPropertyDoubleArray3d 类型,可以得到一个由 3 个 double 值组成的集合,该集合往往用来表示一个(X, Y, Z)值
APT_DoubleArray4d	该 AssetProperty 是 AssetPropertyDoubleArray4d 类型,可以得到一个由 4 个 double 值组成的集合,该集合往往用来表示一个颜色 RGBA 值
APT_Double44	该 AssetProperty 是 APT_Double44 类型,可以得到一个由 4×4 的 double 值组成的矩阵集合,该集合往往用来表示一个颜色坐标变换矩阵
APT_FloatArray	该 AssetProperty 是 AssetPropertyFloatArray 类型,可以得到一个由 float 值组成的集合
APT_List	该 AssetProperty 是 AssetPropertyList 类型。当一个 AssetProperty 的值需要是由多个同类型值的集合表达时,该 AssetProperty 可能被设计为 AssetPropertyList,如由五个 double 值构成的集合。有趣的是,当该集合只有 3 个 double 值时,从 API 中得到的类型可能是 AssetPropertyDoubleArray3d 类型
APT_Reference	该 AssetProperty 是 AssetPropertyReference 类型。这是一个特殊的 AssetProperty 类型,该类型的 Value 值是没有用处的,但是它必定定义了到内链属性集 Asset 的链接,可以通过 GetConnectedProperty 方法来得到
APT_Asset	该 AssetProperty 是 Asset 类型,这个是链接到另一个 Asset 的另一个方法
APT_Properties	该 AssetProperty 是 AssetProperties 类型,可以得到一个 AssetProperty 的集合

注意:AssetProperty 类中的内链属性集(GetConnectedProperty 方法获得)并不一定只当类型为 APT_Reference 时才存在,它的存在与否和 Revit® 内部材质库的定义有关,所以建议对每个 AssetProperty 的实例都通过 NumberOfConnectedProperties 属性来查询是否有内链属性集。

12.5 材料的物理和热度信息

图 12-6 示例了“物理”页的信息，而图 12-7 示例了“热度”页的信息。



图 12-6 材料的物理



图 12-7 材料的热度

把材料的“物理”页和“热度”页列在一起介绍，是因为这两者在 API 中有非常相近的访问方式。

和“外观”页类似，该“物理”页和“热度”页的信息在 Revit® 中也被封装成了一个可被文档中所有材料访问的元素（在 API 中，这两者都是对应了 PropertySetElement 类），材料通过连接该元素来定义材料的物理属性和热度属性。可以通过 Material 类中的 StructuralAssetId 属性来读写物理属性的元素；ThermalAssetId 属性来读写热度属性元素。而 Material 类中的 SetMaterialAspectByPropertySet()方法提供了一个统一的设置物理和热度属性元素的方法。

从 PropertySetElement 类中，通过 GetStructuralAsset()方法来得到一个“物理”属性集合（储存在 StructuralAsset 类中）；或者通过 GetThermalAsset()方法来得到一个“热度”属性集合（储存在 ThermalAsset 类中）。需要注意的是，一个 PropertySetElement 的实例，要么是用来储存物理属性，要么是用来储存热度属性，只能是也必须是其中之一。这从 PropertySetElement 类的构造函数可以了解到。

访问 StructuralAsset 类和 ThermalAsset 类中的属性很简单，直接调用类中的属性来进行读写信息即可。

代码片段 12-3 示例了如何创建材料的物理和热度：Revit SDK 2015\Software DevelopmentKit\Samples\CompoundStructure\CS\Command.cs

代码片段 12-3:创建材料的物理和热度

```
/// < summary>
/// Create a new brick material
/// < /summary>
/// < returns> The specific material< /returns>
private Material CreateSampleBrickMaterial()
{
    SubTransaction createMaterial = new SubTransaction(this.m_document.Document);
    createMaterial.Start();
    Material materialNew = null;

    //Try to copy an existing material.  If it is not available, create a new one.
    Material masonry_Brick = GetMaterial("Brick, Common");
    if (masonry_Brick ! = null)
    {
        materialNew = masonry_Brick.Duplicate(masonry_Brick.Name + "_new");
        System.Diagnostics.Debug.WriteLine(masonry_Brick.MaterialClass);
        materialNew.MaterialClass = "Brick";
    }
    else
    {
        ElementId idNew = Material.Create(m_document.Document, "New Brick Sample");
        materialNew = m_document.Document.GetElement(idNew) as Material;
        materialNew.Color = new Autodesk.Revit.DB.Color(255, 0, 0);
    }
    createMaterial.Commit();

    SubTransaction createPropertySets = new SubTransaction(this.m_document.Document);
    createPropertySets.Start();

    //Create a new structural asset and set properties on it.
    StructuralAsset structuralAsssetBrick = new StructuralAsset("BrickStructuralAsset" ,
    Autodesk.Revit.DB.StructuralAssetClass.Generic);
    structuralAsssetBrick.DampingRatio = .5;

    PropertySetElement pseStructural = PropertySetElement.Create(m_document.Document,
    structuralAsssetBrick);


    //Create a new thermal asset and set properties on it.
     ThermalAsset thermalAssetBrick = new ThermalAsset(" BrickThermalAsset", Autodesk.
     Revit.DB.ThermalMaterialType.Solid);
    thermalAssetBrick.Porosity = 0.1;
    thermalAssetBrick.Permeability = 0.2;
    thermalAssetBrick.Compressibility = .5;
    thermalAssetBrick.ThermalConductivity = .5;

    //Create PropertySets from assets and assign them to the material.
     PropertySetElement pseThermal = PropertySetElement.Create(m_document.Document,
     thermalAssetBrick);
```

```
    createPropertySets.Commit();
    SubTransaction setPropertySets =  new SubTransaction(this.m_document.Document);
    setPropertySets.Start();
    materialNew.SetMaterialAspectByPropertySet(MaterialAspect.Structural, pseStructural.
    Id);
    materialNew.SetMaterialAspectByPropertySet(MaterialAspect.Thermal, pseThermal.Id);

    //also try
    //materialNew.ThermalAssetId =  pseThermal.Id;

    setPropertySets.Commit();
    return materialNew;
}
```

12.6 材料的设置

材料在建筑里是一个很重要的基本概念，建筑里的各种物体都是由各种不同材料组合而成的。而且材料在建筑模型的结构分析，热量分析和可视化上面都有很重要的应用。本节将介绍材料是如何在 Revit® 中被设置到物体上的呢。

材料设置到物体可以大体分为下面几个方式：族类别的预定义材料，系统族的材料设置，构件的材料设置，面的材料设置。

12.6.1 组类别的预定义材料

组类别是 Revit® 中的一个基本特性，几乎所有的模型元素都是属于某一个组类别的，顶层的组类别下可以有分支的组类别。例如，墙是属于“墙”这个组类别，而“墙”组类别下面有预制的“公共边”、“墙饰条-檐口”和“隐藏线”三个分支的组类别。可以设置一个材料作为组类别的预定义材料(图 12-8)。

墙	1	4	黑色	实线	默认墙
公共边	1	4	黑色	实线	
墙饰条 - 檐口	1	3	黑色	实线	默认墙
隐藏线	2	2	黑色	实线	

图 12-8

可以通过 Category 类的 Material 属性来把材料设置到一个组类别上面。

12.6.2 墙等复合结构的材料设置

墙体是由很多层复合构成的综合体，而每层都设置属于该层的材料。在 UI 界面中，要设置层结构的可以在“类型属性”对话框中，单击“结构”属性的“编辑”，将在“类型属性”对话框中打开“编辑部件”对话框。在 API 中，墙体的结构编辑位于 CompoundStructure 类中，可以从 HostObjAttributes. GetCompoundStructure()方法来获取该类的实例，而墙体的类型元素(WallType)继承于 HostObjAttributes 类。同样继承于该类的类型元素(例如：楼板的类型元素(FloorType)和天花板的类型元素(CeilingType)等)也可以进行复合结构的材料设置。

代码片段 12-4 示例了如何为墙体设置材料。

代码片段 12-4:为墙体设置材料

```
public void GetAndSetMaterialForWall(Document doc, Wall wall, Material newMat)
{
    //得到墙体的复合结构(CompoundStructure)实例
    WallType wallType =  wall.WallType;
    CompoundStructure wallCS =  wallType.GetCompoundStructure();

    //得到墙体第一层材料的元素 Id
    CompoundStructureLayer firstLayer =  wallCS.GetLayers().First< CompoundStructureLayer>
    ();
    ElementId currentMatId =  firstLayer.MaterialId;

    //为墙体第一层设置一个新材料
    firstLayer.MaterialId =  newMat.Id;
}
```

12.6.3　楼梯等复杂结构的材料设置

楼梯在 Revit® 中是一个较为特殊的复杂结构,它是由很多独立的元素关联组合起来的,例如,楼梯有很多的扶手,有很多的梯面或者梯段。而扶手是可以自定义的,可以定义出各种各样的扶手,等等。而这些都是可以设置材料的,显然材料的设置不会直接位于楼梯这个复杂结构的组合体上,而是位于各个细节的具体组成元素上。

12.6.4　构件的材料设置

在族文档编辑器中为构件的材料进行定义和预制默认值。具体的步骤如下:

- 在族编辑器中创建的几何图元自带一个材料属性。
- 几何图元的材料属性可以绑定到族参数上(详细介绍参见“第 7 章族(Family)”)。
- 在族文档加载到工程文档后,族参数转化为构件的实例参数和类型参数,可以通过设置相关参数来设置构件的材料。

12.6.5　面的材料设置

Revit® 中提供“填色”和“删除填色”的功能(图 12-9),允许把材料应用于或者取消应用于图元的面。API 中提供了三个方式实现了面的材料设置的功能。一个用于直接关联面和材料,一个用于关联面和材料类型的族参数,最后一个用于删除关联,见表 12-2。

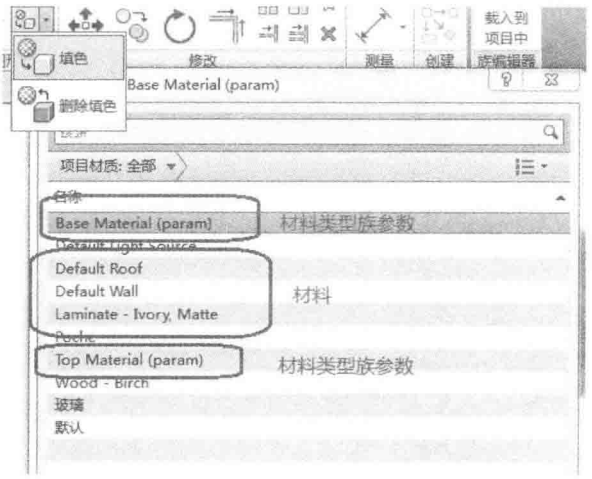


图 12-9　填色材料

表 12-2　　Autodesk. Revit. DB. Document 类中进行面的材料设置的函数

函数名	功能
void Paint(ElementId elementId, Face face, ElementId materialId);	直接关联面和材料
void Paint (ElementId elementId, Face face, FamilyParameter familyParameter);	关联面和材料类型的族参数
void RemovePaint(ElementId elementId, Face face);	删除面和材料或者材料类型族参数的关联

第 13 章　水暖电建模

水暖电建模是 Revit® 应用中非常重要的部分，而且水暖电的设计工作比较繁琐，使用二次开发能够大大提高生产效率，所以相应的 API 就显得尤其重要。本章将介绍风管/管道（Duct/Pipe），管件（MechanicalFitting）、设备（MechanicalEquipment）、水暖电系统（MEPSystem）相应 API 的一些方法，以及使用 API 组建水暖电系统。

13.1　风管/管道（Duct/Pipe）

风管、水管和电线都是继承自 MEPCurve，MEPCurve 继承自 Element，类图见图 13-1。

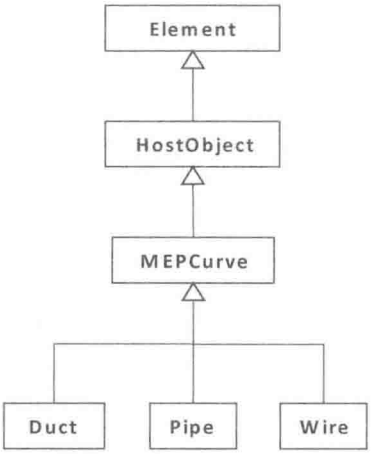


图 13-1　MEPCurve 类图

13.1.1　风管（Duct）

在 Revit® 中，风管用于连接管件，风道末端和机械设备。风管从类型上可分为三类：一般风管，软风管，风管占位符；从形状上分也可分为三类：椭圆管，方管，圆管。Revit® 提供了相应的 API 函数用于创建每种类型的风管。

13.1.2　风管属性和方法

风管的父类 MEPCurve 有以下属性，见表 13-1。

表 13-1　　父类 MEPCurve 属性参数及含义

参数	含义
MEPSystem	管道对应的系统
ConnectorManager	管道电气连接件管理类，通过它可以获取所有的电气连接件
Diameter/Height/Width	获取管道的直径或者高和宽，如果它是圆形风管，那么调用 Heigh/Width 会抛出异常，如果是方形风管，调用 Diameter 就会抛出异常

风管本身有以下重要属性，见表 13-2。

表 13-2　　风管属性参数及含义

参数	含义
DuctType	获取管道的类型
IsPlaceholder	是否是管道占位符

注意：使用 Duct. Location 属性，将它转型成 LocationCurve，再通过 LocationCurve. Curve 属性即可获得管道对应的位置曲线。

13.1.3　创建一般风管

调用 Revit® 的 API 函数 Duct. Create 就可以创建一般风管了。

Duct Create(Document document, ElementId systemTypeId, ElementId ductTypeId,

ElementId levelId, XYZ firstPoint, XYZ secondPoint)

参数含义见表 13-3。

表 13-3　　函数 Duct.Create 的参数含义

参数	含义
Document	文档
systemTypeId	水暖电系统族的 Id
ductTypeId	管道类型的 Id
levelId	管道所在标高的 Id
firstPoint	起点
secondPoint	终点

代码片段 13-1 示例了如何过滤出标高、系统类型和风管类型等参数，以及使用 Duct.Create 创建一个风管。结果见图 13-2。

代码片段 13-1：创建风管

```
public static Duct CreateDuct(Document doc)
{
    ElementId systemTypeId, ductTypeId, levelId;
    systemTypeId = ductTypeId = levelId = ElementId.InvalidElementId;

    //获取标高 Id
    var levelFilter = new ElementClassFilter(typeof(Level));
    FilteredElementCollector levels = new FilteredElementCollector(doc);
    levels = levels.WherePasses(levelFilter);
    foreach (Level level in levels)
    {
        if (level.Name = = "Level 1")
        {
            levelId = level.Id;
            break;
        }
    }
    if(levelId = = ElementId.InvalidElementId)
        throw new Exception("无法标高");

    //获取类型为 SupplyAir 的系统类型
    var systemTypeFilter = new ElementClassFilter(typeof(MEPSystemType));
    FilteredElementCollector systemTypes = new FilteredElementCollector(doc);
    systemTypes = systemTypes.WherePasses(systemTypeFilter);
    List< MEPSystemType> systypes = new List< MEPSystemType> ();
    foreach (MEPSystemType element in systemTypes)
    {
        if (element.SystemClassification = = MEPSystemClassification.SupplyAir)
        {
            systemTypeId = element.Id;
            break;
        }
```

```
    }
    if (systemTypeId = = ElementId.InvalidElementId)
        throw new Exception("无法找到系统类型");

    //获取风管类型
    var ductTypeFilter = new ElementClassFilter(typeof(DuctType));
    FilteredElementCollector ductTypes = new FilteredElementCollector(doc);
    var result = ductTypes.WherePasses(ductTypeFilter).ToList();
    foreach (DuctType element in result)
    {
        ductTypeId = element.Id;
        break;
    }

    //创建风管
    using (Transaction transaction = new Transaction(doc))
    {
        transaction.Start("创建风管");
        Duct duct = Duct.Create(doc, systemTypeId, ductTypeId, levelId, new XYZ(0, 10, 0),
        new XYZ(10, 0, 0));
        transaction.Commit();
        return duct;
    }
}
```



图 13-2　创建风管

13.1.4　创建软风管

软风管的创建方法和一般风管的创建方法基本一致，也是通过给定某些点来创建。调用 Revit® 的 API 函数 FlexDuct.Create 就可以创建软风管了。它有两个重载函数：

- FlexDuct Create (Document document, ElementId systemTypeId, ElementId ductTypeId, ElementId levelId, XYZ startTangent, XYZ endTangent, IList＜XYZ＞ points)；
- FlexDuct Create (Document document, ElementId systemTypeId, ElementId ductTypeId, ElementId levelId, IList＜XYZ＞ points)。

创建软风管的方式与创建风管类似，区别是：软风管使用 FlexDuctType 作为风管类型；软风管可以指定多个拐点，并可以指定起点和终点的拐角（使用 startTangent 和 endTangent 指定方向）。

13.1.5 创建风管占位符

风管占位符并不表示物理风管，只是说明在相应的位置上会放置相应的一般风管。调用 Revit® 的 API 函数 Duct. CreatePlaceholder 就可以创建风管占位符了。

Revit® 的 API 提供了将风管占位符转换成风管的函数：MechanicalUtils. ConvertDuctPlaceholders。

13.1.6 管道(Pipe)

在 Revit® 中，管道用于连接管件、管路附件和卫浴装置等。与风管类似，管道也可分为三类：一般管道、软管道、管道占位符。Revit® 提供了相应的 API 函数用于创建每种类型的管道。

管道有和风管类似的属性，请参见风管的属性和方法。

用户可以调用 Revit® 的 API 函数 Pipe. Create 创建一般的管道，使用 FlexPipe. Create 来创建软管，使用 Pipe. CreatePlaceholder 来创建占位管道。代码示例请参见风管的创建。

13.2 电气连接件(Connector)

MEP 的元素都是通过电气连接件进行连接。使用 Duct. ConnectorManager. Connectors 或者 FamilyInstance. MEPModel. ConnectorManagers. Connectors 可以获取管道或者 MEP 模型元素的所有电气连接件，图 13-3 给出了两种电气连接件的例子。

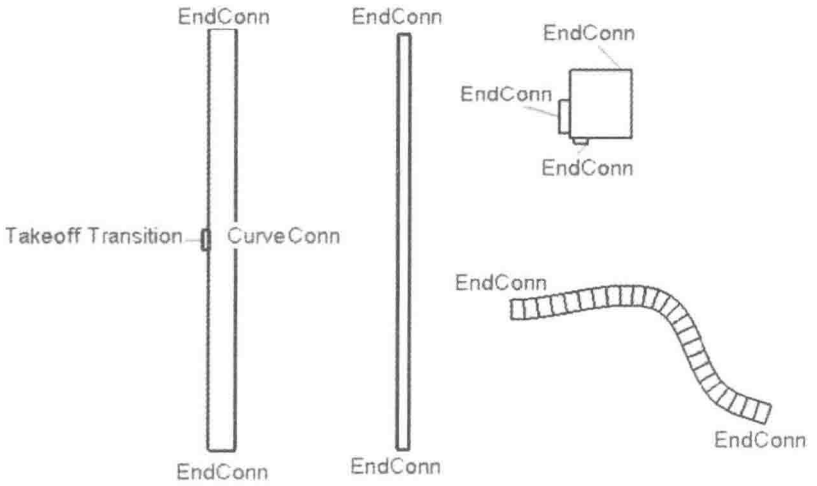


图 13-3 电气连接件

代码片段 13-2 展示如何获取电气连接件的 owner 及电气连接件连接了什么。

代码片段 13-2：

```
public void GetElementAtConnector(Connector connector)
{
   MEPSystem mepSystem =  connector.MEPSystem;
   if (null ! =  mepSystem)
   {
      string message =  "Connector is owned by: " +  connector.Owner.Name;
```

```
        if (connector.IsConnected = = true)
        {
            ConnectorSet connectorSet = connector.AllRefs;
            ConnectorSetIterator csi = connectorSet.ForwardIterator();
            while (csi.MoveNext())
            {
                Connector connected = csi.Current as Connector;
                if (null ! = connected)
                {
                    // look for physical connections
                    if (connected.ConnectorType = = ConnectorType.EndConn ||
                        connected.ConnectorType = = ConnectorType.CurveConn ||
                        connected.ConnectorType = = ConnectorType.PhysicalConn)
                    {
                        message + = "\nConnector is connected to: " + connected.Owner.Name;
                        message + = "\nConnection type is: " + connected.ConnectorType;
                    }
                }
            }
        }
        else
        {
            message + = "\nConnector is not connected to anything.";
        }
        MessageBox.Show(message, "Revit®");
    }
}
```

运行后的对话框，见图 13-4。

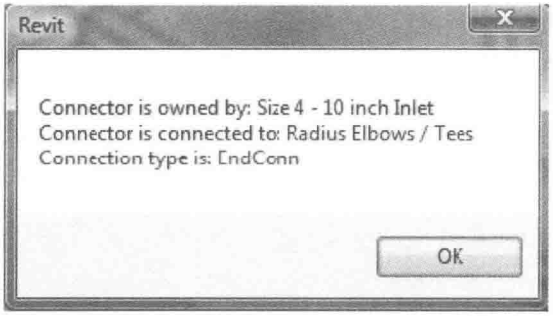


图 13-4　获取电气连接件的 Owner 和连接对象

13.3　水暖电模型(MEPModel)

水暖电模型(MEPModel)是管件(MechanicalFitting)或者设备(MechanicalEquipment)的父类，要获取管件或者设备的水暖电模型，首先过滤出 FamilyInstance，然后使用 FamilyInstance. MEPModel 属性获得 MEPModel，最后再把 MEPModel 转型相应的子类，类图见图 13-5。

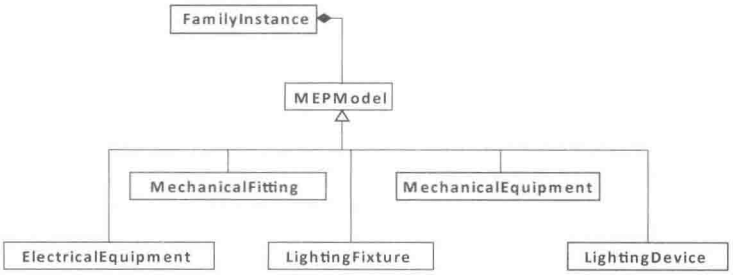


图 13-5　MEPModel 类图

管件中包括风管管件(duct fitting)和管件(pipe fitting),通常是用于连接不同的风管或管道。管件基本可以分为弯头型、T 字型及十字型三种,分别用于连接两根、三根和四根风管或管道。Revit® API 有两种方式可以创建风管,一种是使用 Document.Create 下面相应的函数,另一种是使用 MechanicalUtils 和 PlumbingUtils 类的连接相关的函数。

创建管件主要包括下面五个方法,分别对应创建弯头(Elbow)、三通(Tee)、过渡件(Transition)、管接头(Union)和四通(Cross),返回值是 FamilyInstance。参数都是用来创建管件对应电气连接件,例如,创建弯头将需要两个连接件,创建三通需要三个。

- NewElbowFitting(Connector connector1, Connector connector2)
- NewTeeFitting (Connector connector1, Connector connector2, Connector connector3)
- NewTransitionFitting(Connector connector1, Connector connector2)
- NewUnionFitting(Connector connector1, Connector connector2)
- NewCrossFitting (Connector connector1, Connector connector2, Connector connector3, Connector connector4)

代码片段 13-3 示例了如何创建弯头连接两个风管。结果见图 13-6。

代码片段 13-3:创建弯头

```
public static void ConnectTwoDuctsWithElbow(Document doc, Duct duct1, Duct duct2)
{
    double minDistance = double.MaxValue;
    Connector connector1, connector2;
    connector1 = connector2 = null;

    //找到距离最近的两个 Connector
    foreach (Connector con1 in duct1.ConnectorManager.Connectors)
    {
        foreach (Connector con2 in duct2.ConnectorManager.Connectors)
        {
            var dis = con1.Origin.DistanceTo(con2.Origin);
            if (dis < minDistance)
            {
                minDistance = dis;
```

```
                connector1 = con1;
                connector2 = con2;
            }
        }
    }
    if (connector1 != null && connector2 != null)
    {
        using (Transaction transaction = new Transaction(doc))
        {
            transaction.Start("Create Elbow");
            doc.Create.NewElbowFitting(connector1, connector2);
            transaction.Commit();
        }
    }
}
```

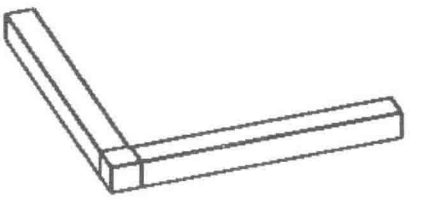


图 13-6　创建弯头

13.4　水暖电系统(MEPSystem)

水暖电系统类 MEPSystem 也是 Element。类图见图 13-7。

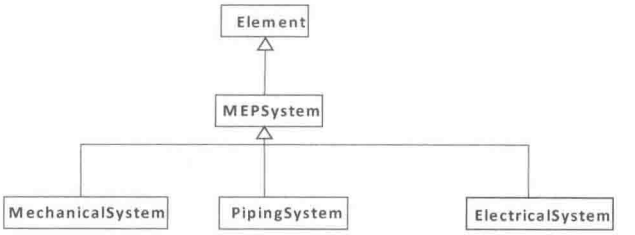


图 13-7　MEPSystem 类图

MEPSystem 的子类有 MechanicalSystem，PipeingSystem，ElectricalSystem，分别对应风管系统，水管系统和电器系统。

风管系统通常是为了便于计算风管中送风流向和流量而设定的一个逻辑概念。只要将风道末端和机械设备放置在工程中就可以创建补给、返回和废气系统用于连接各个不同的部件。

水管系统是为了便于计算设备的流量和流速而设定的一个逻辑概念。水管系统独立于工程文件中的水管，仅用于显示各个卫浴装置间连接方式和水流方向。只要将卫浴装置放

置在工程中就可以创建家用热水、家用冷水及卫浴系统来连接各个不同的装置。

13.4.1 水暖电系统的创建

MEP 系统的创建方式基本相同，下面仅以风管系统为例说明。

13.4.2 水暖电系统属性和方法

MEPSystem 本身的几个重要的属性见表 13-4。

表 13-4 MEPSystem 的属性及含义

属性	含义
BaseEquipment	系统的设备
BaseEquipmentConnector	系统设备的电气连接件
Elements	系统除设备以外的其他构件

重要的方法见表 13-5。

表 13-5 MEPSystem 的方法及含义

方法	含义
Add(ConnectorSet connectors)	往系统添加构件
Remove(ICollection<ElementId> elementIds)	通过 ElementId 移除构件
Remove(ConnectorSet connectors)	通过电气连接件移除构件

13.4.3 创建风管系统

创建风管系统，使用 Document. Create 对象的方法：

• MechanicalSystem NewMechanicalSystem (Connector baseEquipmentConnector, ConnectorSet connectors, DuctSystemType ductSystemType)

函数参数的含义见表 13-6。

表 13-6 函数 Document. Create 的参数含义

参数	含义
baseEquipmentConnector	设备电气连接件
connectors	系统构件上的电气连接件
ductSystemType	管道系统的类型，可以是送风、回风等

代码片段 13-4 示例了用 API 在一个在机械设备和风管终端间创建风管系统的简单例子。结果见图 13-8。

代码片段 13-4：在机械设备和风管终端间创建风带系列

```
public static void CreateMechanicalSystem (Document doc, DuctSystemType systemType,
    FamilyInstance equipment, params FamilyInstance[] airTerminals)
{
    var me = equipment.MEPModel as MEPModel;
    if (me != null)
    {
        //获取符合条件的设备上的电气连接件
        var connsOfEquipment = GetConnector(me.ConnectorManager, systemType).ToList();
```

```
        if (connsOfEquipment.Count > 0)
        {
            //选取设备上的第一个电气连接件
            var equipmentConn = connsOfEquipment[0];

            var connsOfTerminals = new ConnectorSet();
            foreach (var terminal in airTerminals)
            {
                //获取符合条件的散流器上的电气连接件,取第一个
                var airTerminalEquipment = terminal.MEPModel as MEPModel;
                var airConns = GetConnector(airTerminalEquipment.ConnectorManager,
                systemType);
                var airConn = airConns.FirstOrDefault();
                if (airConn != null)
                    connsOfTerminals.Insert(airConn);
            }

            //创建风管系统
            using (Transaction transaction = new Transaction(doc))
            {
                transaction.Start("Create System");
                doc.Create.NewMechanicalSystem(equipmentConn, connsOfTerminals,
                systemType);
                transaction.Commit();
            }
        }
    }
}

//获取符合管道系统的电气连接件
private static IEnumerable<Connector> GetConnector(ConnectorManager conMgr,
    DuctSystemType systemType)
{
    foreach (Connector conn in conMgr.Connectors)
    {
        // Domain 为 Hvac,否则 DuctSystemType 属性会抛出异常
        if (conn.Domain == Domain.DomainHvac && conn.DuctSystemType == systemType)
        {
            yield return conn;
        }
    }
}
```

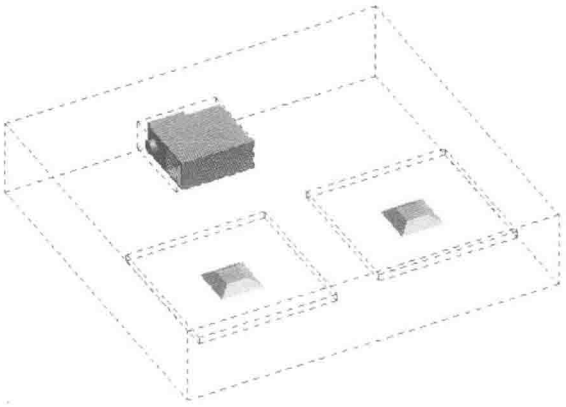


图 13-8　创建风管系统结果

13.5　水暖电设置

水暖电设置菜单在“管理”→“MEP 设置”菜单下，见图 13-9。

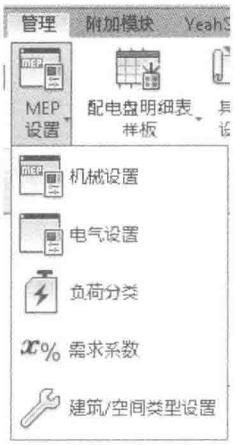


图 13-9　水暖电设置菜单

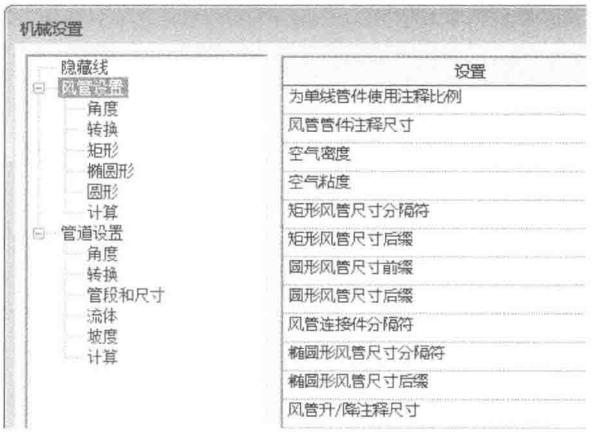


图 13-10　机械设置

表 13-7 列出了各种 MEP 设置对应的类。

表 13-7　　水暖电设置相关类

设置	对应的类
电气设置	ElectricalSetting
机械设置	DuctSettings，PipeSettings
机械设置里面的矩形，椭圆形，圆形尺寸设置	DuctSizeSettings
负荷分类	ElectricalLoadClassification，获取所有的负荷分类可以使用 ElementClassFilter
需求系数	ElectricalDemandFactorDefinition，获取所有的需求系数可以使用 ElementClassFilter

界面上的机械设置见图 13-10。

水暖电设置类的类图见图 13-11。

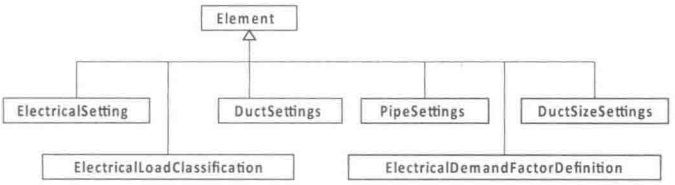


图 13-11　水暖电设置类类图

获取设置类的方式有两种：

① 通过他们本身的静态函数。例如，获取 DuctSizeSettings，可以使用静态函数：DuctSizeSettings. GetDuctSizeSettings(Document aDoc)

② 通过 ElementClassFilter 过滤出所有的设置元素，如：ElectricalLoadClassification，ElectricalDemandFactorDefinition。

13.5.1　风管设置

在 API 中，风管设置类可以用于设置诸如风管管件注释尺寸、空气密度黏度、连接件分隔符等参数。代码片段 13-5 演示了使用 API 变更风管设置中的角度参数。

代码片段 13-5：使用 API 变更风管设置中的角度参数

```
using (Transaction tran =  new Transaction(doc, "DuctSettings"))
{
tran.Start();
   //获取 Revit 的风管设置对象
   DuctSettings ductSettings =  DuctSettings.GetDuctSettings(doc);
//设置风管设置的角度参数
   ductSettings.FittingAngleUsage =  FittingAngleUsage.UseAnAngleIncrement;
   tran.Commit();
}
```

结果如图 13-12 所示，可以看到“角度”参数被设置成了“设置角度增量”。

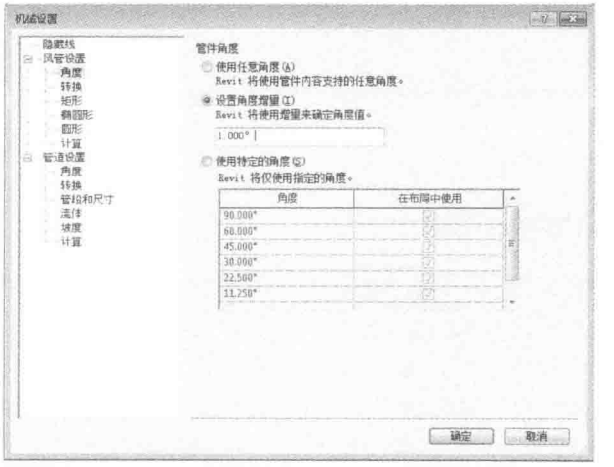


图 13-12　角度增量设置

其他参数的设置也可以参考代码片段 13-5 进行。

13.5.2 管道设置

和风管设置类似，可以通过 API 对管道的各个参数进行设置。

13.5.3 电气设置

和风管设置、管道设置类似，我们可以通过 API 对电气相关的各个参数进行设置。

代码片段 13-6 演示了使用 API 变更电气设置中的一些参数。

代码片段 13-6：添加电压

```
using (Transaction tran =  new Transaction(doc, "ElectricalSetting"))
{
tran.Start();
   //获取 Revit 的电气设置对象
   ElectricalSetting electricalSettings =  ElectricalSetting.GetElectricalSettings(doc);
   //向电压定义参数中加入 220 伏电压
electricalSettings.AddVoltageType("220 伏电压", 220, 220, 220);
tran.Commit();
}
```

结果如图 13-13 所示，可以看到“电压定义”参数中加入了“220 伏电压”。

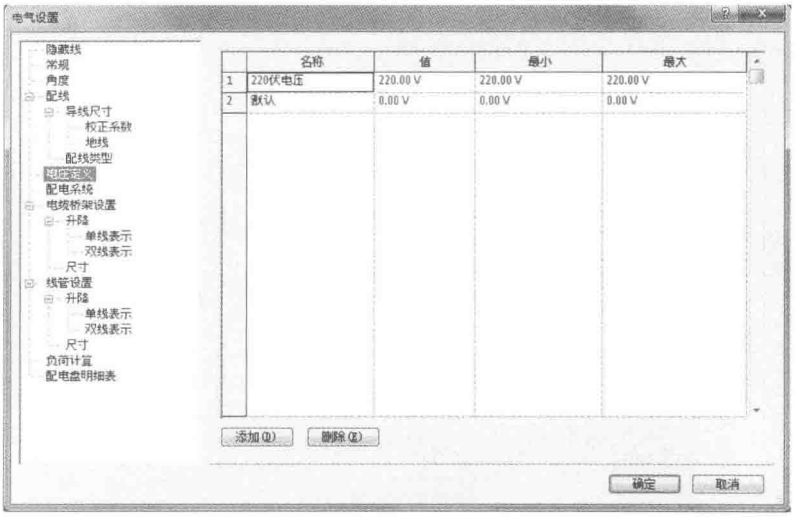


图 13-13 添加电压

13.6 空间和分区(Space/Zone)

在 Revit® 中，空间是专门用于在 MEP 中分析容积的组件。空间组件中维护了一些参数表示空间组件所在区域的区域信息，这些信息会被用于计算热负荷和冷负荷。

空间能被放置在工程中也可以取消放置及从工程中删除。取消放置和删除是不一样的。空间刚被加进模型时会被放置到一个默认的分区中。空间会在剖面视图中显示。在正视图和 3D 图中不能放置空间，在这两个视图中空间也不会被显示。空间组件应该被放置

在未被占用的分区中。

分区组件由一个或多个空间组件组成，通常用来表示这些空间都有一个共通的环境或设计要求。只要空间组件还未被占用，这个空间组件就可以被添加到分区中。在不同层高上的空间组件也可以被添加到同一个分区中。

代码片段 13-7 演示了怎样使用 API 添加空间到模型中并将其添加到一个分区内。模型中已创建了一些墙，这些墙围出了几个封闭区域，见图 13-14。

图 13-14　封闭区域

代码片段 13-7：创建分区和空间

```
using (Transaction tran =  new Transaction(doc, "Create new space"))
{
tran.Start();
   Level level1 =  doc.GetElement(new ElementId(311)) as Level;
   Phase phase1 =  doc.GetElement(new ElementId(86961)) as Phase;
  //用时期(Phase)创建空间
   Space space1 =  doc.Create.NewSpace(phase1);
   //用标高(Level)和平面的点创建空间
   Space space2 =  doc.Create.NewSpace(level1, new UV(- 31, - 19));
   //用时期(Phase)，标高(Level)和平面的点创建空间
   Space space3 =  doc.Create.NewSpace(level1, phase1, new UV(4, 16));
//用标高(Level)和时期(Phase)创建分区
Zone zone1 =  doc.Create.NewZone(level1, phase1);
SpaceSet spaceSet =  new SpaceSet();
   spaceSet.Insert(space1);
   spaceSet.Insert(space2);
spaceSet.Insert(space3);
//将空间添加到分区中
   zone1.AddSpaces(spaceSet);
   tran.Commit();
}
```

运行代码结果见图 13-15，可以看到空间被添加到了两块封闭分区中。利用 API 创建空间时，系统会根据函数调用时给定的点自动找出匹配的封闭分区来添加空间。创建出的三个空间也被添加同一个分区内。

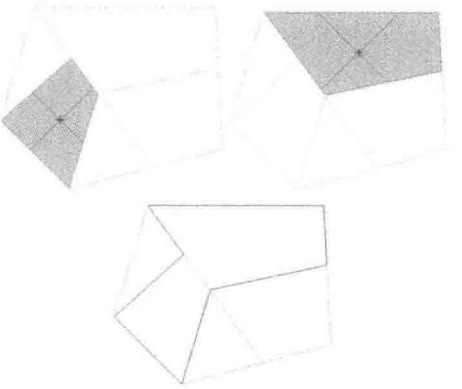


图 13-15　创建分区和空间

第 14 章　宏(Macro)

首先,来回答这样一个问题:“什么是宏,为什么要使用宏?”宏是一种程序,旨在通过实现重复任务的自动化来节省你的时间。每个宏可执行一系列预定义的步骤来完成特定任务。这些步骤应该是可重复执行的,操作是可预见的。

例如,可以定义宏,用于向项目添加轴网、旋转选定对象,或者收集有关结构中所有房间的面积的信息。其他一般示例包括:

- 定位 Revit® 内容并将其提取到外部文件;
- 优化几何图形或参数;
- 创建多种类型的图元;
- 导入和导出外部文件格式。

宏也是基于 Revit® API 来编写的。本章将介绍标准 Revit® API 和 Revit® 宏使用的 API 之间的区别,但是提醒经验丰富的开发人员注意:无需通过. addin 来注册宏,也无需手动将 Revit® API. dll 添加为一个引用(已经自动添加)。

14.1　Revit® 宏简介

14.1.1　宏管理器

宏管理器是一个用户界面,可用于:

- 启动 Revit® 宏 IDE,在该 IDE 中可以添加、编辑、构建和调试宏;
- 从分类列表中运行先前构建的宏。

宏管理器对话框如图 14-1 所示。这些选项卡指示宏的范围或级别。

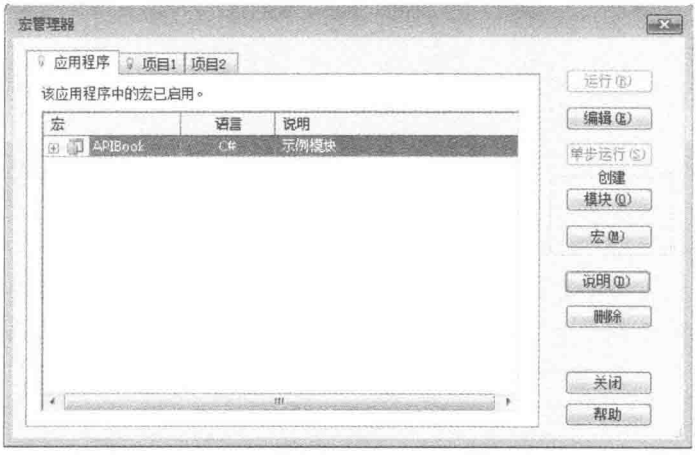


图 14-1　宏管理器

- “应用程序”选项卡。“应用程序”选项卡列出了可用于 Revit® 应用程序当前实例中所有打开的 Revit® 项目的宏模块。它始终是左侧第一个选项卡且处于活动状态(由 图标指示)。该选项卡始终可用,无论项目是否处于打开状态.

注意:如果将 RVT 文件发送给另一台计算机的用户,则应用程序级宏将不可用。

- 活动文档选项卡。该活动文档选项卡表示 Revit® 中当前处于活动状态的文档。在本例中,文档不一定包含内嵌宏,尽管它可以包含。该选项卡具有活动文档的名称(在本例

中为“项目 1”)且由 图标指示。如果文档未打开,则该选项卡不可见。

• 非活动文档选项卡。非活动文档选项卡表示包含内嵌宏的已打开文档(请参见下文)。该选项卡具有文档的名称(在本例中为“项目 2”)。如果非活动文档没有包含内嵌宏则不行显示该文档的选项卡。

可以使用这些选项卡添加、修改、构建和删除模块及宏。

14.1.2 宏和模块

模块是对宏的分组,实际上是一个编程语言的工程项目。宏在运行时在模块中可以是独立的,也可以彼此之间共享代码或实用程序。这些宏按照在其父模块下组织的方式在宏管理器中进行排列(图 14-2)。

模块中的宏是一起显示和构建的。表示该模块的图标还会显示其当前状态。表 14-1 列出了各个图标的含义。

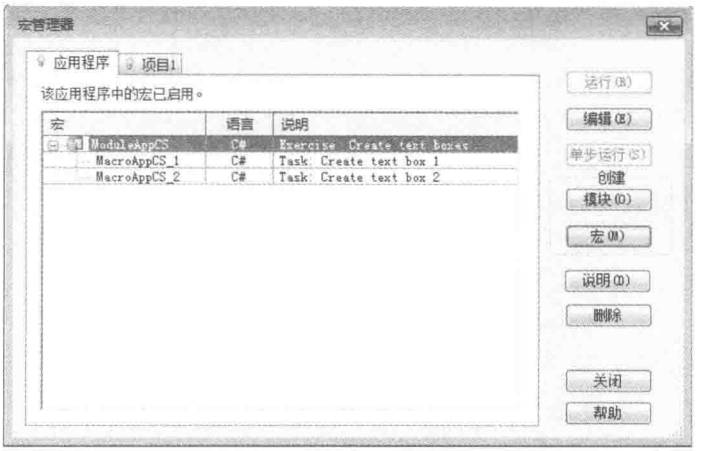


图 14-2 模块和宏

表 14-1 宏图标含义

图标	宏状态	图标	宏状态
	该模块已成功构建、启用并载入。此模块已准备好运行。请参见在宏管理器中运行宏		模块无法构建。请参见在 Revit® 宏 IDE 中构建宏
	该模块已被编辑,但未构建。请参见在 Revit® 宏 IDE 中构建宏		模块无法载入
	模块已被禁用。请参见宏安全性		模块已损坏

14.1.3 应用程序级和文档级模块

基本上,应用程序级宏创建后即可在几乎任何文档中使用。此外,它们可以自行运行,而不要求在 Revit® 中打开项目。这样一来,就可以灵活地执行下列操作:

- 向 Revit® 中添加工具;
- 在文档打开时对其进行修改;
- 批量打开文档;
- 将新标准或设置应用于新文档。

如果实施这些用途,最好创建应用程序级宏。这些宏还可以启动宏所需的事务来对文档进行修改。

文档级宏工具是为特定项目编写的,且保存在 Revit® 文档中。

14.1.4　Revit® 宏实现语言

开发者可以使用实现语言 C#，VB.NET，Ruby 或 Python 在 Revit® 中创建宏，用户所做的选择将决定在 Revit® 宏 IDE 中生成和编辑的源代码模板的类型。

14.1.5　宏项目文件位置

使用 Revit® 宏 IDE 时，只有已成功保存和构建的宏才会显示在宏管理器的分类列表中。在讨论载入 Revit® 宏到宏 IDE 中显示初始代码示例之前，先介绍一下宏项目文件在计算机上的保存位置。

应用程序级的宏项目存储在 Applicatioin Data 目录下：

- Windows XP：C:\Documents and Settings\All Users\Application Data\Autodesk\Revit\Macros\<版本>\<产品>\AppHookup
- Windows 7：C:\ProgramData\Autodesk\Revit\Macros\<版本>\<产品>\AppHookup

文档级宏项目存储在 RVT 文件中。在磁盘上，如果打开了关联的 RVT 项目，则任何已构建和保存的宏都将临时存储下来(在 Temp 目录下)。但是，当这些文档级宏文件对应的 Revit® 项目文档(.rvt)关闭时，这些文件将被从本地计算机中删除。

在介绍了整个工具和流程之后，下面介绍具体的任务。

14.2　Revit® 宏开发的基本工作流程

下面是宏的整个创建和使用工作流程：

① 单击“管理”选项卡➤“宏”面板(宏管理器)。

② 根据应用程序级设计或文档级设计创建模块。

③ 使用 Revit® 宏 IDE 为带有实施代码的模块定义宏。

④ 构建模块及其宏。

⑤ 在宏管理器中运行宏以观察结果。

14.2.1　创建模块

在宏管理器中通过以下步骤来创建一个模块。

注意：如果应用程序或当前文档已包含模块，则必须在“宏安全性”设置中启用宏。请参见本章宏安全性一节。

① 单击“管理”选项卡➤“宏”面板(宏管理器)。

② 选择“应用程序”选项卡(对于应用程序级宏)或“项目”选项卡(对于文档级宏)。

③ 在“宏管理器”对话框的“创建”部分中，单击“模块”。“创建新模块”对话框将打开，见图 14-3。

④ 输入下列信息：

- 在“模块名称”字段中输入名称。
- 在“语言”字段中，选择“C#”，“VB.NET”，“Ruby”或“Python”。
- 在“说明”字段中添加模块的简要说明。此项为可选项。

⑤ 单击“确定”。

新模块将被创建并列在宏管理器中，见图 14-4。Revit® 宏 IDE 也会随之打开。现在可以添加一个或多个宏。



图 14-3　采用 C# 创建应用程序级模块



图 14-4　新创建的宏

新模块是通过相对应编程语言的模板来创建的。其中已经设置好了许多必要的属性。

例如，在应用程序级宏的 C # 模板中，Revit® 宏 IDE 已经：

- 包括了必需的 using 指令。
- 标识了模块名称空间(在上图中为 ModuleAppCS)。
- 添加了 ThisApplication 类定义。
- 添加了 Module_Startup() 和 Module_Shutdown() 的方法。



图 14-5　宏 IDE 显示新创建的模块

此外，需注意 Revit® 宏 IDE 项目浏览器还会显示上下文，见图 14-5。

14.2.2　创建宏

有两种方法来创建一个宏。第一种是通过宏管理器来创建：



图 14-6　创建新宏对话框

① 在“宏管理器”中，选择“应用程序”选项卡(对于应用程序级宏)或“项目”选项卡(对于文档级宏)，这就是放置宏的位置。

② 在“宏管理器”对话框的“创建”部分中，单击“宏”。“创建新宏”对话框将打开(图 14-6)。

③ 输入下列信息：

- 在“宏名称”字段中输入名称，只支持英文。
- 从“宏位置”列表中选择一个父模块。该父模块的语言将定义宏的语言。
- (可选)在“说明”字段中添加宏的简要说明。

④ 单击“确定”。新的宏将被放置在宏管理器的父模块中(图 14-7)。



图 14-7　宏管理器显示新创建的宏

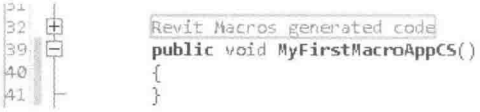


图 14-8　宏 IDE 显示新创建的宏

同时，Revit® 宏 IDE 应用程序会随之启动并自动加入了相应的宏函数；或者在已经打开的情况下，自动加入宏函数。然后就可以在 Revit® 宏 IDE 中写宏的实现代码。

⑤ 在 IDE 的主窗口中，现在可以输入源代码，见图 14-8。

代码片段 14-1 创建一个文字注释元素。

代码片段 14-1：创建一个文字注释元素

```
public void MyFirstMacroAppCS()
{
  Autodesk.Revit.DB.XYZ baseVec =  Application.Create.NewXYZ(1.0, 0.0, 0.0);
  Autodesk.Revit.DB.XYZ upVec =  Application.Create.NewXYZ(0.0, 0.0, 1.0);
  Autodesk.Revit.DB.XYZ origin =  Application.Create.NewXYZ(0.0, 0.0, 0.0);

  Autodesk.Revit.DB.TextAlignFlags align =  Autodesk.Revit.DB.TextAlignFlags.TEF_ALIGN_
  LEFT | Autodesk.Revit.DB.TextAlignFlags.TEF_ALIGN_TOP;

  string strText =  "My First Macro, App level, C# !";
  double lineWidth =  4.0 / 12.0;

  Autodesk.Revit.DB.View pView =  ActiveUIDocument.Document.ActiveView;
  Autodesk.Revit.DB.Transaction t =  new Autodesk.Revit.DB.Transaction(ActiveUIDocument.
  Document, "NewTextNote");
  t.Start();
  ActiveUIDocument.Document.Create.NewTextNote(pView, origin, baseVec, upVec, lineWidth,
  align, strText);

  t.Commit();
}
```

⑥ 单击“生成”➤“生成解决方案”。如果代码没有问题，Revit® 宏 IDE 的左下角会显示“生成成功”。

⑦ 重复步骤①到⑥来添加其他宏。

第二种添加宏的方法比较直接，就是直接在打开的 Revit® 宏 IDE 中写宏函数和其实现

代码，写完编译通过即可。一个正确的宏需要满足以下条件：

① 宏声明在 ThisApplication 或 ThisDocument 类中。

② 宏函数访问权限是 public 的。

③ 宏函数的返回值是 void。

④ 宏函数无输入参数。

具体可以参见下节的宏示例代码。

14.2.3 宏示例代码

下面列出了 C#，VB.NET，Ruby 和 Python 四种语言应用程序级和文档级的示例代码。需要注意的是，应用程序级的 this 或 self 相对应 Revit® API 中的 UIApplication，而文档级的 this 和 self 则对应于 Revit® API 中的 UIDocument。

C#应用程序级宏代码示例见代码片段 14-2。

代码片段 14-2：创建一个文字注释元素(C#)

```
public void MyFirstMacroAppCS()
{
  Autodesk.Revit.DB.XYZ baseVec = Application.Create.NewXYZ(1.0, 0.0, 0.0);
  Autodesk.Revit.DB.XYZ upVec = Application.Create.NewXYZ(0.0, 0.0, 1.0);
  Autodesk.Revit.DB.XYZ origin = Application.Create.NewXYZ(0.0, 0.0, 0.0);

  Autodesk.Revit.DB.TextAlignFlags align = Autodesk.Revit.DB.TextAlignFlags.TEF_ALIGN_
  LEFT | Autodesk.Revit.DB.TextAlignFlags.TEF_ALIGN_TOP;

  string strText = "My First Macro, App level, C# !";
  double lineWidth = 4.0 / 12.0;

  Autodesk.Revit.DB.View pView = ActiveUIDocument.Document.ActiveView;
  Autodesk.Revit.DB.Transaction t = new Autodesk.Revit.DB.Transaction(ActiveUIDocument.
  Document, "NewTextNote");
  t.Start();
  ActiveUIDocument.Document.Create.NewTextNote(pView, origin, baseVec, upVec, lineWidth,
  align, strText);

  t.Commit();
}
```

VB.NET 应用程序级宏代码示例见代码片段 14-3。

代码片段 14-3：创建一个文字注释元素(VB.NET)

```
Public Sub MyFirstMacroAppVB()
    Dim baseVec As Autodesk.Revit.DB.XYZ = Application.Create.NewXYZ(1.0, 0.0, 0.0)
    Dim upVec As Autodesk.Revit.DB.XYZ = Application.Create.NewXYZ(0.0, 0.0, 1.0)
    Dim origin As Autodesk.Revit.DB.XYZ = Application.Create.NewXYZ(0.0, 0.0, 0.0)
    Dim align As Autodesk.Revit.DB.TextAlignFlags = Autodesk.Revit.DB.TextAlignFlags.TEF
    _ALIGN_LEFT Or Autodesk.Revit.DB.TextAlignFlags.TEF_ALIGN_TOP
    Dim strText As String = "My First Macro, App Level, VB.NET!"
```

```
    Dim lineWidth As Double = 4.0 / 12.0
    Dim pView As Autodesk.Revit.DB.View = ActiveUIDocument.Document.ActiveView
    Dim Transaction As Autodesk.Revit.DB.Transaction = New Autodesk.Revit.DB.Transaction
    (ActiveUIDocument.Document, "NewTextNote")
    Transaction.Start()
      ActiveUIDocument. Document. Create. NewTextNote ( pView, origin, baseVec, upVec,
    lineWidth, align, strText)
    Transaction.Commit()
End Sub
```

Ruby 应用程序级宏代码示例见代码片段 14-4。

代码片段 14-4:显示版本和文件标题(Ruby)

```
def SampleMacroImplementation()
    revitVersion = self.Application.VersionBuild
    docTitle = ""
    if (self.ActiveUIDocument.Document! = nil)
      docTitle = self.ActiveUIDocument.Document.Title
    end
    TaskDialog.Show("Ruby Sample", "Revit Version: " + revitVersion + ", Document title: " +
    docTitle)
end
```

Python 应用程序级宏代码示例见代码片段 14-5。

代码片段 14-5:显示版本和文件标题(Python)

```
def SampleMacro(self):
    revitVersion = self.Application.VersionBuild
    docTitle = ""
    if (self.ActiveUIDocument.Document! = None):
      docTitle = self.ActiveUIDocument.Document.Title;
    TaskDialog.Show("Python Sample", "Revit Version: " + revitVersion + ", Document title: "
    + docTitle)
```

C# 文档级宏代码示例见代码片段 14-6。

代码片段 14-6:创建一个文字注释元素(C#)

```
public void MyFirstMacroDocCS()
{
    Autodesk.Revit.DB.XYZ baseVec = Document.Application.Create.NewXYZ(0.0, 0.0, 1.0);
     Autodesk.Revit.DB.XYZ upVec = Document.Application.Create.NewXYZ(0.0, 0.0, 1.
    0);
     Autodesk.Revit.DB.XYZ origin = Document.Application.Create.NewXYZ(0.0, 0.0,
    0.0);
    Autodesk.Revit.DB.TextAlignFlags align = Autodesk.Revit.DB.TextAlignFlags.TEF_ALIGN_
    LEFT | Autodesk.Revit.DB.TextAlignFlags.TEF_ALIGN_TOP;
```

```
    string strText = "My First Macro, Doc level, C# !";
    double lineWidth = 4.0 / 12.0;
     Autodesk.Revit.DB.Transaction t = new Autodesk.Revit.DB.Transaction(Document, "
     NewTextNote");
    t.Start();
    Autodesk.Revit.DB.View pView = Document.ActiveView;
    Document.Create.NewTextNote(pView, origin, baseVec, upVec, lineWidth, align, strText);
    t.Commit();
}
```

VB.NET 文档级宏代码示例见代码片段 14-7。

代码片段 14-7:创建一个文字注释元素(VB.NET)

```
Public Sub MyFirstMacroDocVB()
    Dim baseVec As Autodesk.Revit.DB.XYZ = Document.Application.Create.NewXYZ(1.0, 0.0, 0.
    0)
    Dim upVec As Autodesk.Revit.DB.XYZ = Document.Application.Create.NewXYZ(0.0, 0.0, 1.
    0)
    Dim origin As Autodesk.Revit.DB.XYZ = Document.Application.Create.NewXYZ(0.0, 0.0,
    0.0)
    Dim align As Autodesk.Revit.DB.TextAlignFlags = Autodesk.Revit.DB.TextAlignFlags.TEF
    _ALIGN_LEFT Or Autodesk.Revit.DB.TextAlignFlags.TEF_ALIGN_TOP
    Dim strText As String = "My First Macro, Doc Level, VB.NET!"
    Dim lineWidth As Double = 4.0 / 12.0
    Dim pView As Autodesk.Revit.DB.View = Document.ActiveView
    Dim Transaction As Autodesk.Revit.DB.Transaction = New Autodesk.Revit.DB.Transaction
    (Document, "NewTextNote")
    Transaction.Start()
    Document.Create.NewTextNote(pView, origin, baseVec, upVec, lineWidth, align, strText)
    Transaction.Commit()
End Sub
```

Ruby 文档级宏代码示例见代码片段 14-8。

代码片段 14-8:显示版本和文件标题(Ruby)

```
def SampleMacroImplementation()
    revitVersion = self.Application.Application.VersionBuild
    docTitle = ""
    if (self.Document! = nil)
    docTitle = self.Document.Title
    end
    TaskDialog.Show("Ruby Sample", "Revit Version:" + revitVersion + ", Document title:" +
    docTitle)
end
```

Python 应用程序级宏代码示例见代码片段 14-9。

代码片段 14-9:显示版本和文件标题(Python)

```
def SampleMacro(self):
    revitVersion =  self.Application.Application.VersionBuild
    docTitle =  ""
    if (self.Document! =  None):
      docTitle =  self.Document.Title;
    TaskDialog.Show("Python Sample", "Revit Version: " +  revitVersion +  ", Document title: "
    +  docTitle)
```

14.3　修改与删除模块和宏

在“宏管理器”中,选择要修改的宏的“应用程序”选项卡(对于应用程序级宏)或“项目”选项卡(对于文档级宏)。

14.3.1　模块

要删除模块,请将其选中并单击“删除”。删除模块会永久删除该模块及其关联的宏。

要修改模块,请将其选中并单击“编辑”。IDE 将打开该模块,并且你可以编辑该模块及其关联的宏。

14.3.2　宏

要删除宏,请将其选中并单击“删除”。删除宏不会将其永久删除,而是会在源代码中将其注释掉。要永久删除宏,需要在 IDE 中对其进行编辑来手动删除。

要修改宏,请将其选中并单击“编辑”。IDE 将打开该宏,并且你可以编辑该宏、其模块及其关联的宏。

14.4　在宏管理器中运行宏

在宏管理器中运行宏的步骤如下:

① 单击“管理”选项卡➤“宏”面板➤(宏管理器)。

② 选择一个宏。

③ 单击“运行”。

也双击“宏管理器”中的各个宏以运行它们。

要运行宏还必须注意相应的宏安全性的设置。请参见本章宏安全性一节。

14.5　调试宏

可以使用“宏管理器”中的“单步运行”选项来调试宏。

在“宏管理器”中,选择分类列表中你所需的宏,然后单击“单步运行”。IDE 将进入调试模式并停止位于第一行中的宏。

- 按 F10 键逐行分步执行宏代码。
- 按 F5 键或“继续”,可移动下一个断点(可以在宏中有选择地设置断点。在代码窗口的左边距中单击,或者在代码中单击,然后按 F7 键或在“调试”菜单中单击“设置断点”。断点将显示为红色的圆形。)。

注意:在使用"单步运行"时,请避免切换 Windows 用户或允许 Windows 进入休眠模式。在某些情况下,这可能会导致错误和意外丢失数据。

在 Revit® 宏 IDE 中,可使用"监视"、"局部变量"和其他选项执行调试任务,如检查变量值。

如果调试"Module_Startup"和"Module_Shutdown",必须先打开 Revit® 宏 IDE,然后手动设置好断点,再选择需要调试的宏,点击"单步运行"。这样,在调试加入该宏之前,先会在 Module_Startup 的断点处停止。

14.6 宏安全性

默认情况下,可以选择启用和禁用宏。这些设置是对整个应用程序级宏或文档级宏起作用的。这可以对用户的工作和计算机起到保护作用,避免意外运行危险的恶意代码。处理宏时,应提防因宏的弱点而带来的风险,这一点非常重要。应仅运行从信赖的地方获取的宏。

1) 设置应用程序级宏安全性

① 禁用应用程序级宏可以限制 Revit® 启动时运行应用程序级宏,并且"宏管理器"中的"运行"和"单步运行"工具将无法使用。但仍可以查看、编辑和构建代码,在修改后不会改变当前模块状态。

② 单击"管理"选项卡➤"宏"面板➤"宏安全性"。

③ 选择"宏"选项卡。

在"应用程序宏安全性设置"部分中,选择"启用应用程序宏"或"禁用应用程序宏"。

④ 单击"确定"。

⑤ 关闭并重新打开 Revit®。

注意:对应用程序级宏设置执行的任何修改都将在重新启动 Revit® 后生效。该设置将持久存在,直到下次被修改为止。

2) 设置文档级宏安全性

① 禁用文档级宏可以限制 Revit® 打开文档时运行文档级的宏,并且任何已打开项目的"宏管理器"中的"运行"和"单步运行"工具将无法使用。开发者将仍然可以查看、编辑和构建代码,但修改后不会改变当前模块状态。

单击"管理"选项卡➤"宏"面板➤"宏安全性"。

② 选择"宏"选项卡。

③ 在"文档宏安全性设置"部分中,选择下列选项之一:

• 启用文档宏前询问。这是默认设置。宏被禁用,但如果在打开 Revit® 项目时宏存在,系统会提示启用宏。然后,可以选择在检测到宏时启用宏。

• 禁用文档宏。在打开项目时禁用文档级宏。要在此状态下启用宏,必须关闭项目,调整"宏安全性"设置,并重新打开项目。

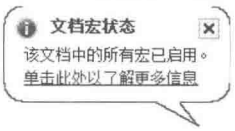
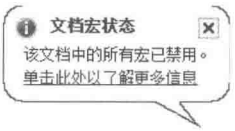
• 启用文档宏。在打开项目时启用文档级宏。要在此状态下禁用宏,必须关闭项目,调整"宏安全性"设置,并重新打开项目。

应该尽可能避免启用文档宏设置。系统不会提示何时打开关于宏以及可运行的潜在的危险恶意代码的文档。如果不熟悉 Revit® 宏或经常打开来自不熟悉的来源的项目,建议使

用“启用文档宏前询问”。只有当知道内嵌宏的来源时，才应使用“启用文档宏设置”。

为了帮助开发者确定宏安全性的当前状态，系统会在 Revit® 应用程序框架的右下角提供状态图标和通知提示。表 14-2 列出了各种通知的图示。

表 14-2　　文档级宏状态和警告

宏安全性设置	状态图标	打开时通知
启用文档宏		
禁用文档宏		
启用文档宏前询问	用户定义	一个警告对话框将提示启用或禁用宏。产生的答案将显示正确的图标。这是默认设置

14.8　标准 Revit® API 与 Revit® 宏使用的 API 的区别

标准 Revit® API 的功能基本都可以被 Revit® 宏使用，但也有一些区别。表 14-3 汇总了标准 Revit® API 和 Revit® 宏 API 之间的区别。

表 14-3　　标准 Revit® API 和 Revit® 宏 API 之间的区别

特性或功能	标准 Revit® API	Revit® 宏 API
声明	必须实现 IExternalCommand 接口及其 Execute 方法	在 ThisApplication 或 ThisDocument 类中声明不带参数的公共方法和空返回类型
Application 对象	通过 externalCommandData. Application 访问 Application 对象	C# 和 VB. NET 中的 Application 关键字指向应用程序级宏的 Application 对象。对于文档级宏，Document. Application 指向 Application 对象
功能	API 外部应用程序可以通过一个外部应用程序来为每个外部命令创建 RibbonPanels、RibbonItem、PulldownButton 和 PushButton	不支持

第 15 章　其他语言（VB. NET，C++/CLI，F#）

除了 C# 程序设计语言，Revit® 插件还支持 VB. NET、C++/CLI 和 F# 等托管程序设计语言。本章会简单介绍如何用 VB. NET、C++/CLI 和 F# 来写一个 Revit® 插件。

15.1　VB. NET

这节将介绍如何用 VB. NET 语言来写一个 Hello Revit 的插件。该插件作为一个 Revit® 外部命令，执行时弹出一个提示框显示“Hello Revit!”。

首先，创建一个 VB. NET 的类库工程，创建步骤和 C# 的工程类似，参考 Revit® API 基础一章。然后，添加 Revit API. dll 和 Revit APIUI. dll 的引用。通过工程属性中的引用页面来添加，见图 15-1。

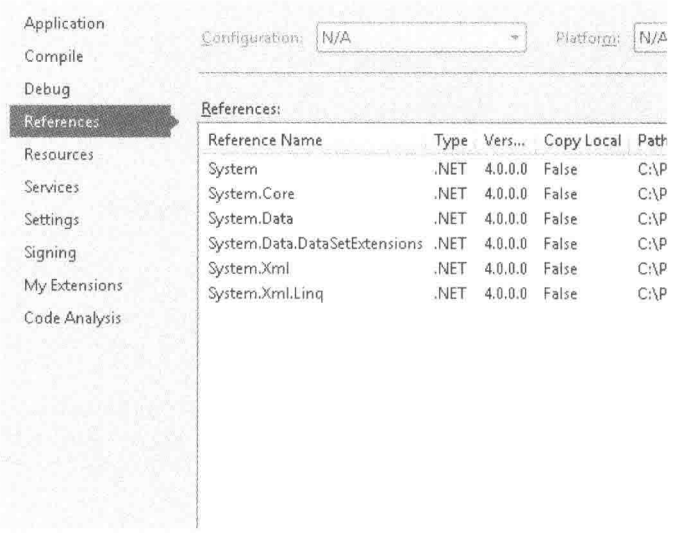


图 15-1　VB. NET 添加引用

然后将默认生成的 Class1 类继承并实现接口 IExternalCommand。见代码片段 15-1。

代码片段 15-1：HelloRevit（VB. NET）

```
Imports System

Imports Autodesk
Imports Autodesk.Revit.DB
Imports Autodesk.Revit.UI
Imports Autodesk.Revit.ApplicationServices

< Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMode.Manual)> _
```

```
Public Class Command
    Implements Autodesk.Revit.UI.IExternalCommand

    Public Function Execute(ByVal commandData As Autodesk.Revit.UI.ExternalCommandData, _
        ByRef message As String, ByVal elements As Autodesk.Revit.DB.ElementSet) _
        As Autodesk.Revit.UI.Result Implements Autodesk.Revit.UI.IExternalCommand.Execute

        Dim app As Application
        app =  commandData.Application.Application
        Dim activeDoc As Document
        activeDoc =  commandData.Application.ActiveUIDocument.Document
        Dim mainDialog As TaskDialog
        mainDialog =  New TaskDialog("Hello, Revit!")
        mainDialog.MainInstruction =  "Hello, Revit!"
        mainDialog.MainContent =  _
                "This sample shows how a basic ExternalCommand can be added to the Revit user
                interface." _
                    +  " It uses a Revit task dialog to communicate information to the
                    interactive user.\n"
        mainDialog.Show()
        Return Autodesk.Revit.UI.Result.Succeeded
    End Function
End Class
```

通过 Visual Studio 创建成功后，提供一个. addin 文件来注册该外部命令，见代码片段 15-2。

代码片段 15-2：HelloRevit. addin 文件

```
< ? xml version= "1.0" encoding= "utf-8"? >
< RevitAddIns>
< AddIn Type= "Command">
< Assembly> HelloRevit.dll< /Assembly>
< ClientId> 0e6385db-b2c0-4397-8702-b1b889bad37c< /ClientId>
< FullClassName> HelloRevit.Class1< /FullClassName>
< VendorId> ADSK< /VendorId>
< /AddIn>
< /RevitAddIns>
```

有关 Addin 文件的详细信息请参见本书第 3 章插件集成。

安装完成后，启动 Revit®，运行“外部工具”下拉列表中的“HelloRevit”，执行后将弹出一个对话框。

有关插件的调试和 C# 类似。

15.2　C++/CLI

这节将介绍如何用 C++/CLI 语言来写一个 Hello Revit 的插件。该插件作为一个

Revit®外部命令，执行时弹出一个提示框显示“HelloRevit!”。

与 VB. NET 一样，首先也要创建一个类库工程，语言选择 Visual C++下的 CLR。添加 Revit API. dll 和 Revit APIUI. dll 作为引用。通过工程属性的通用属性来添加，见图 15-2。

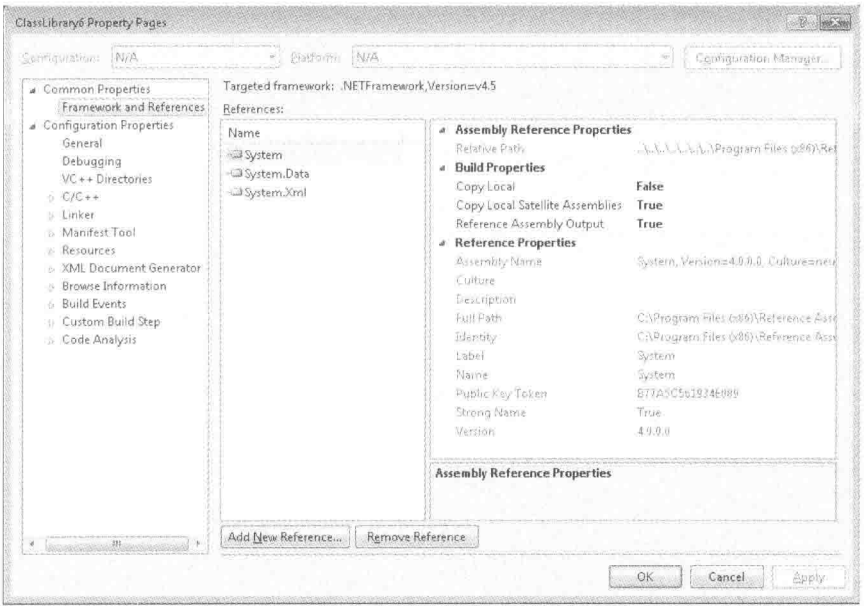


图 15-2　C++/CLI 添加引用

然后将默认生成的 Class1 类继承并实现接口 IExternalCommand，见代码片段 15-3 和代码片段 15-4。

代码片段 15-3：HelloRevit. h

```
# pragma once

using namespace System;

using namespace Autodesk::Revit;
using namespace Autodesk::Revit::DB;
using namespace Autodesk::Revit::UI;
using namespace Autodesk::Revit::ApplicationServices;

namespace HelloRevit {

    public ref class Class1 : IExternalCommand
    {
    public:
        virtual Result Execute (ExternalCommandData^ commandData, String^% message,
        ElementSet^ elements);
    };
}
```

代码片段 15-4：HelloRevit. cpp

```
# include "stdafx.h"

# include "HelloRevit.h"

namespace HelloRevit
{
      Result  Class1:: Execute ( ExternalCommandData^  commandData,  String^ %  message,
      ElementSet^ elements)
    {
      Application^ app =  commandData-> Application-> Application;
      Document^ activeDoc =  commandData-> Application-> ActiveUIDocument-> Document;
      TaskDialog^ mainDialog =  gcnew TaskDialog("Hello, Revit!");
      mainDialog-> MainInstruction =  "Hello, Revit!";
      mainDialog-> MainContent =  "This sample shows how a basic ExternalCommand can be added
      to the Revit user interface. It uses a Revit task dialog to communicate information to
      the interactive user.";
      mainDialog-> Show();
      return Autodesk::Revit::UI::Result::Succeeded;
    }
}
```

通过 Visual Studio 创建成功后，提供一个. addin 文件来注册该外部命令，见代码片段 15-5。

代码片段 15-5：HelloRevit. addin 文件

```
< ? xml version= "1.0" encoding= "utf-8"? >
< RevitAddIns>
< AddIn Type= "Command">
< Assembly> HelloRevit.dll< /Assembly>
< ClientId> 0e6385db-b2c0-4397-8702-b1b889bad37c< /ClientId>
< FullClassName> HelloRevit.Class1< /FullClassName>
< VendorId> ADSK< /VendorId>
< /AddIn>
< /RevitAddIns>
```

有关 Addin 文件的详细信息请参见本书第 3 章插件集成。

安装完成后，启动 Revit®，运行“外部工具”下拉列表中的“HelloRevit”，执行后将弹出一个对话框。

有关插件的调试和 C# 类似。

15.3　F#

F# 是由微软发展的为微软. NET 语言提供运行环境的程序设计语言。这节将介绍如何用 F# 语言来写一个“HelloRevit”的插件。该插件作为一个 Revit® 外部命令，执行时弹出一个提示框显示“HelloRevit!”。

与前面介绍的程序设计语言类似，我们先要在 Visual Studio 里新建一个类库工程，语言选择 Visual F# 下的类库。添加 RevitAPI. dll 和 RevitAPIUI. dll 作为引用。通过右键点击引用，添加引用来添加。然后将写一个 Class1 类继承并实现接口 IExternalCommand，见代码片段 15-6。

代码片段 15-6：HelloRevit. h

```
module Module1

open Autodesk.Revit
open Autodesk.Revit.UI
open Autodesk.Revit.Attributes
open Autodesk.Revit.DB

[< Transaction(TransactionMode.Manual)> ]
type Class1 =
    interface Autodesk.Revit.UI.IExternalCommand with
        member this.Execute(commandData,
                            message : string byref,
                            elements) =
            let uiApp =  commandData.Application
            let app   =  uiApp.Application
            let uiDoc =  uiApp.ActiveUIDocument
            let doc   =  uiDoc.Document

            use mainDialog =  new TaskDialog("Hello, Revit!")
            mainDialog.MainInstruction < - "This sample shows how a basic ExternalCommand
            can be added to the Revit user interface. It uses a Revit task dialog to
            communicate information to the interactive user."
            let result =  mainDialog.Show()
            Autodesk.Revit.UI.Result.Succeeded
```

通过 Visual Studio 创建成功后，提供一个. addin 文件来注册该外部命令，见代码片段 15-7。

代码片段 15-7：HelloRevit. addin 文件

```
< ? xml version= "1.0" encoding= "utf-8"? >
< RevitAddIns>
< AddIn Type= "Command">
< Assembly> HelloRevit.dll< /Assembly>
< ClientId> 1585c811-3dc4-43f4-9ec5-e7e695a6ff72< /ClientId>
< FullClassName> HelloRevit.Class1< /FullClassName>
< VendorId> ADSK< /VendorId>
< /AddIn>
< /RevitAddIns>
```

有关 Addin 文件的详细信息请参见本书第 3 章插件集成。

安装完成后，启动 Revit®，运行“外部工具”下拉列表中的“HelloRevit”，执行后将弹出一个对话框。

有关插件的调试和 C# 类似。